QUEST SOFTWARE®

JProbe® 8.3

User Guide



© 2009 Quest Software, Inc. ALL RIGHTS RESERVED.

This guide contains proprietary information protected by copyright. The software described in this guide is furnished under a software license or nondisclosure agreement. This software may be used or copied only in accordance with the terms of the applicable agreement. No part of this guide may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording for any purpose other than the purchaser's personal use without the written permission of Quest Software, Inc.

If you have any questions regarding your potential use of this material, contact:
Quest Software World Headquarters
LEGAL Dept
5 Polaris Way
Aliso Viejo, CA 92656
www.quest.com
email: legal@quest.com

Refer to our Web site for regional and international office information.

JProbe Patents

Patent pending.

Trademarks

Quest, Quest Software, the Quest Software logo, AccessManager, ActiveRoles, Aelita, Akonix, AppAssure, Benchmark Factory, Big Brother, BusinessInsight, ChangeAuditor, ChangeManager, DeployDirector, DirectoryAnalyzer, DirectoryTroubleshooter, DS Analyzer, DS Expert, ERDisk, Foglight, GPOADmin, Imceda, IntelliProfile, InTrust, Invirtus, iToken, I/Watch, JClass, Jint, JProbe, LeccoTech, LiteSpeed, LiveReorg, LogADmin, MessageStats, Monosphere, NBSpool, NetBase, NetControl, Npulse, NetPro, PassGo, PerformaSure, Quest Central, Quest vToolkit, Quest vWorkSpace, ReportADmin, RestoreADmin, SelfServiceADmin, SharePlex, Sitraka, SmartAlarm, Spotlight, SQL LiteSpeed, SQL Navigator, SQL Watch, SQLab, Stat, StealthCollect, Storage Horizon, Tag and Follow, Toad, T.O.A.D., Toad World, vAutomator, vControl, vConverter, vFoglight, vOptimizer Pro, vPackager, vRanger, vRanger Pro, vSpotlight, vStream, vToad, Vintela, Virtual DBA, VizionCore, Vizioncore vAutomation Suite, Vizioncore vBackup, Vizioncore vEssentials, Vizioncore vMigrator, Vizioncore vReplicator, Vizioncore vTraffic, Vizioncore vWorkflow, WebDefender, Webthority, Xaffire, and XRT are trademarks and registered trademarks of Quest Software, Inc in the United States of America and other countries. Other trademarks and registered trademarks used in this guide are property of their respective owners.

Disclaimer

The information in this document is provided in connection with Quest products. No license, express or implied, by estoppel or otherwise, to any intellectual property right is granted by this document or in connection with the sale of Quest products. EXCEPT AS SET FORTH IN QUEST'S TERMS AND CONDITIONS AS SPECIFIED IN THE LICENSE AGREEMENT FOR THIS PRODUCT, QUEST ASSUMES NO LIABILITY WHATSOEVER AND DISCLAIMS ANY EXPRESS, IMPLIED OR STATUTORY WARRANTY RELATING TO ITS PRODUCTS INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. IN NO EVENT SHALL QUEST BE LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, PUNITIVE, SPECIAL OR INCIDENTAL DAMAGES (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF PROFITS, BUSINESS INTERRUPTION OR LOSS OF INFORMATION) ARISING OUT OF THE USE OR INABILITY TO USE THIS DOCUMENT, EVEN IF QUEST HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Quest makes no representations or warranties with respect to the accuracy or completeness of the contents of this document and reserves the right to make changes to specifications and product descriptions at any time without notice. Quest does not make any commitment to update the information contained in this document.

License Credits and Third Party Information

See *Third_Party_Contributions.htm* in your JProbe *\doc* installation directory.

User Guide
November 2009

Table of Contents

Introduction to This Guide 15

About JProbe 16

About This Guide 16

JProbe Documentation Suite 18

Core Documentation Set 19

Feedback on the Documentation 19

Text Conventions 20

About Quest Software, Inc. 20

Contacting Quest Software 21

Contacting Quest Support 21

Quest Communities 21

Configuring JProbe to Analyze Your Application 23

Configuration Overview 24

Configuring a Java EE Application 24

Configuring a Java SE Application 27

Defining a JProbe Configuration 29

Specifying Your Java EE Application Server 31

Specifying Your Java Application 31

Main Class and Working Directory 32

Application Arguments 33

Classpath 34

Specifying a Java Virtual Machine 36

Java Executable 36

Java Options 37

Specifying Your Code 38

My Application Filters Table 39

Elements Table 43
Specifying a JProbe Analysis Type 45
Specifying Advanced JProbe Options 46
Advanced JProbe Options 47
JProbe Port # 48
Snapshot Basename 48
Reviewing the Configuration Summary 49
Saving the Configuration 50
Integrating Your Application 51
Saving the JProbe Startup Script 51
Executing the JProbe Startup Script 53
Saving the Configuration and Integrating 55
Running Your Script 55
Running Your Script from JProbe Console 55
Running Your Script from JProbe Configuration Tool 57
Running Your Script and Connecting the Console 58
Managing Configurations 59
Pre-Defined Demo Configurations 59
Copying a Configuration 59
Editing a Configuration 60
Deleting a Configuration 61
Creating a Configuration from a Settings File 61
Renaming a Settings File 63
What Can I Do if My Application Server Is Not Supported? 64
Creating a Configuration for an Unsupported Application Server 65
Requesting a Module from Quest Support 67
Modifying an Application Server Startup Script 68

Starting Your Application with JProbe Attached 71
Running a Java Application with JProbe 72
Running a Java EE Application with JProbe 73
Runtime Notes for Application Servers 75
Apache Geronimo v1.0 76
Apache Geronimo v1.1 77
Apache Geronimo v2.0 77

Apache Geronimo v2.1 78
Apache Tomcat v5.5 79
Apache Tomcat v6.0 80
BEA Systems WebLogic v9.x/8.x 81
BEA Systems WebLogic Portal v9.x 81
BEA Systems WebLogic v10.0 81
IBM WebSphere AS Community Edition v2.0.0.1 82
IBM WebSphere AS Community Edition v2.1 82
IBM WebSphere AS v6.x 83
IBM WebSphere Portal Server v6.x 84
IBM WebSphere AS v7.0 85
JBoss v4.2.x 86
JBoss v5.x 87
Open Source GlassFish v2 UR2 87
Open Source GlassFish v3 88
Open Source Java Service Wrapper v3.3 89
Oracle AS 10g v10.1.3.0 or below 90
Oracle AS 10g v10.1.3.1 or above 91
Oracle WebLogic v10.3 92
Oracle WebLogic Server 11g R1 (v10.3.1) 92
Oracle OC4J 10g v10.0.x 93
Oracle OC4J 10g v10.1.2 or below 93
Oracle OC4J 10g v10.1.3 93
SpringSource dm Server v1.0.2 94
SpringSource dm Server v2.0 94
What to Do When the Console Does Not Connect 95
Connecting a JProbe Console to an Application Started Without JProbe 96
Connecting JProbe to a Local Application Started Without JProbe 96
Connecting JProbe to a Remote Application Started Without JProbe 97

Strategies for Memory Analysis 101
Recording Data 102
Runtime Summary 102
Memory Pools Tab 103
GC Data Tab 104

Heap Data Tab 105

Running a Memory Analysis 107
Finding Memory Leaks 108
Understanding Loitering Objects 108
Setting the Recording Level 110
Performing a Basic Memory Analysis 112
Tuning Garbage Collection and Finding Over-allocations 117
Understanding Garbage Collection Issues 117
Reducing Over-allocation of Objects 119
Examining Garbage Collection Data 120

Learning More About Memory Options 123
Analysis Options for Memory 124
Filters for Memory 124
Triggers for Memory 125
Initial Recording for Memory 126
Importing Heap Dumps 127
Importing an IBM JVM Heap Dump 127
Importing an HPROF Heap Dump 129

Exploring Memory Analysis Views 131
Exploring the Memory Runtime Summary View 132
Understanding the Memory Runtime Summary View 132
Interacting with the Memory Runtime Summary View 135
Exploring the Instances View 138
Opening the Instances View 138
Understanding the Instances View 139
Filtering Data in the Instances View 141
Interacting with the Instances View 144
Exploring the Dominating Instances View 148
Opening the Dominating Instances View 148
Understanding the Dominating Instances View 149
Filtering Data in the Dominating Instances View 151
Interacting with the Dominating Instances View 152
Exploring the Pinned Instances View 155

Opening the Pinned Instances View 155
Understanding the Pinned Instances View 155
Interacting with the Pinned Instances View 157
Exploring the Allocations View 160
Opening the Allocations View 160
Understanding the Allocations View 160
Interacting with the Allocations View 161
Exploring the Allocations Detail View 163
Opening the Allocations Detail View 163
Understanding the Allocations Detail View 163
Interacting with the Allocations Detail View 164
Exploring the Instance Detail View 166
Opening the Instance Detail View 166
Understanding the Instance Detail View 167
Querying a Set of Instances 170
Interacting with the Instance Detail View 171
Exploring the Call Traces View 174
Opening the Call Traces View 175
Understanding the Call Traces View 175
Customizing the Call Traces View 177
Interacting with the Call Traces View 178
Exploring the Merged Allocation Points View 181
Opening the Merged Allocation Points View 181
Understanding the Merged Allocation Points View 181
Interacting with the Merged Allocation Points View 182
Exploring the Heap Graph View 185
Opening the Heap Graph 185
Understanding the Heap Graph 186
Customizing the Heap Graph 187
Interacting with the Heap Graph 187
Exploring the Leak Doctor View 190
Opening the Leak Doctor 190
Understanding the Leak Doctor 192
Interacting with the Leak Doctor View 195
Exploring the Memory Source View 196

Opening the Source View 197
Interacting with the Source View 197
Exploring the Memory Difference View 197
Opening the Memory Difference View 198
Understanding the Memory Difference View 199
Using Filters in the Memory Difference View 200
Interacting with the Memory Difference View 200
Setting Preferences for Memory Views 202

Reporting Memory Analysis Results 203
Setting Report Options 204
Runtime Summary Report 204
Instances Report 204
Dominating Instances Report 205
Pinned Instances Report 205
Allocations Report 206
Allocations Detail Report 206
Instance Detail Report 207
Call Traces Report 207
Merged Allocation Points Report 208
Heap Graph Report 208
Memory Snapshot Difference Report 208

Troubleshooting Memory Analysis Results 211
Memory Troubleshooting Tips 212
Memory Session Runtime Tips 214
Memory Help Messages 215
No Heap Data 215
No Stack Trace Data 216
Operation Not Valid for Categories 216

Strategies for Performance Analysis 217
Strategies for Achieving Program Efficiency 218
Analyzing an Existing Program 218
Focusing on a Known Inefficient Algorithm 220
Understanding the Memory Pools Chart 221

Understanding Performance Data....222
Collecting Data on Methods....222
Building a Calling Context Tree....222
How Data is Affected by Using Performance Features....225
Recursive Methods....232

Running a Performance Analysis....235
Setting Up JProbe for a Performance Analysis Session....236
Setting the Recording Level....236
Running an Analysis Session....237
Exercising Your Program....237
Saving Baseline Snapshots....239
Understanding Data Collection....240
Investigating Your Results....243
Identifying Performance Bottlenecks....243
Investigating Performance Bottlenecks....248
Narrowing Your Investigation....249
Measuring Performance Improvements....258
Identifying Deadlocks....260
Investigating Performance Deadlocks....264

Learning More About Performance Options....265
Analysis Options for Performance....266
Timing Settings for Performance....266
JDBC Component Instrumentation....267
Filters for Performance....267
Automation Settings for Performance....268
Triggers for Performance....271

Exploring Performance Analysis Views....273
Exploring the Runtime Summary View....274
Understanding the Runtime Summary View....274
Interacting with the Performance Runtime Summary View....279
Exploring the Summary View....283
Opening the Summary View....283
Understanding the Summary View....285

Interacting with the Summary View ... 292
Exploring the Methods View ... 297
Opening the Methods View ... 297
Understanding the Methods View ... 297
Interacting with the Methods View ... 305
Customizing the Call Graph ... 323
Exploring the Method Detail View ... 325
Opening the Method Detail View ... 326
Understanding the Method Detail View ... 326
Navigating the Method Detail View ... 332
Customizing the Method Detail View ... 333
Exploring the Performance Source View ... 335
Opening the Source View ... 335
Interacting with the Source View ... 336
Exploring the Performance Difference View ... 338
Opening the Performance Difference View ... 338
Understanding the Performance Difference View ... 339
Interacting with the Performance Difference View ... 340
Setting Preferences for Performance Views ... 342

Reporting Performance Analysis Results ... 343
Setting Report Options ... 344
Runtime Summary Report ... 344
Summary Report ... 345
Methods Report ... 346
Method Detail Report ... 347
Performance Difference Report ... 347

Troubleshooting Performance Analysis Results ... 349
Performance Troubleshooting Tips ... 350
Performance Session Runtime Tips ... 351
Performance Help Messages ... 353
No Line Data ... 353

Strategies for Coverage Analysis ... 355
Assessing an Existing Test Suite ... 356

Determining Overall Coverage 356
Modifying a Test Suite 356
Creating a New Test Suite 356
Keeping Up with Development 357

Running a Coverage Analysis 359
Setting Up JProbe for a Basic Coverage Analysis 360
Running Test Cases with JProbe 361
Investigating Your Results 363

Learning More About Coverage Options 367
Analysis Options for Coverage 368
Coverage Options 368
Filters for Coverage 369
Triggers for Coverage 371

Exploring Coverage Analysis Views 373
Exploring the Coverage Session 374
Controlling a Session with Runtime Toolbar Buttons 374
Understanding the Data in the Coverage Session 375
Interacting with the Coverage Session 376
Exploring the Snapshot Browser 378
Opening the Snapshot Browser 378
Understanding the Data in the Snapshot Browser 378
Interacting with the Snapshot Browser 382
Exploring the Coverage Source View 384
Opening the Source Code 384
Understanding the Data in a Source View 385
Interacting with the Source View 388
Setting Preferences for Coverage Views 390
Displaying Percentages in Bar Charts 391
Filtering Out Catch Block Data 391
Customizing the Colors Used to Display Results 392
Learning More About Coverage Results 393
Nonintuitive Results for Some Source Code 393
Nonintuitive Results for Conditions 396

Reporting Coverage Analysis Results 399
Coverage Session Report 400
Class Summary 400
Class and Method Summaries 400
Snapshot Browser Report 401
Application Summary 402
Package Summary 403
Class Summary 403
Source Data 404
Report Filters 405

Troubleshooting Coverage Analysis Results 409
Coverage Troubleshooting Tips 410

Learning How to Use JProbe Effectively 413
Hints for Success with JProbe 414
Understanding the Architecture for Your Program 414
Selecting Use Cases 414
Compiling Your Program 415
Learning How JProbe Works 416
JProbe Analysis Engine 417
JProbe Connection Manager 417
JProbe Console 417
JProbe Snapshots 418
Demos and Tutorials 418
A Quick Tour of the Interface 419
Configurations Navigator 419
Snapshots Navigator 420

Refining Data Collection and Presentation 421
Refining Data Collection Filters 422
Adding Filters 422
Changing Filter Execution Order 424
Removing Filters 424
Setting Triggers 425

Adding Triggers 425
Setting Advanced Trigger Controls 427
Changing Trigger Execution Order 429
Removing Triggers 429
Refining When a Trigger Executes 429
Adding a Trigger in the Body of a Method 431
Creating and Managing Categories 432
Adding Categories 432
Adding Folders 436
Editing Categories 436
Renaming Categories or Folders 437
Deleting Categories or Folders 437

Communicating Results 439
Generating HTML Reports 440
Printing Data to a PDF File 441
Exporting Table Data to a CSV File 442
Exporting Runtime Chart Data to a CSV File 443
Saving a Chart as an Image 443
Viewing the Execution Log 444
Viewing the Session History 445
Generating a Support Bundle 446

Managing Snapshots 447
Setting the Current Snapshot 448
Saving Snapshots 448
Loading Saved Snapshots 449
Unloading Snapshots 449
Saving Snapshots on Exit 450
Managing Snapshot Transfers 451

Setting Preferences 453
Controlling How JProbe Operates 454
Changing How JProbe Exports to a CSV File 455
Hiding Package Names and Method Signatures 455
Specifying Source File Locations 455

Configuring Memory Pools 456

Running Remote Sessions 459

Starting and Viewing a Remote Session 460

JProbe License Options 463

Node Locked Licensing 464

Requesting Node Locked Licenses 464

Registering Node Locked Licenses 466

Per Seat Licensing 468

Concurrent Licensing 469

Using Concurrent Licenses 469

Enterprise Licensing 472

Index 473

Introduction to This Guide

This chapter provides information about what is contained in the *JProbe User Guide*. It also provides information about the JProbe documentation suite and Quest Software.

This chapter contains the following sections:

About JProbe....16
About This Guide....16
JProbe Documentation Suite....18
Text Conventions....20
About Quest Software, Inc.....20

About JProbe

JProbe is an enterprise-class Java profiler that provides intelligent diagnostics on memory usage, performance, and test coverage. It allows developers to quickly pinpoint and repair the root cause of application code performance and stability problems that obstruct component and integration integrity.

JProbe provides three types of analysis:

- Memory analysis—allows a developer to identify and resolve Java memory leaks and object cycling, to ensure optimal program efficiency and stability.
- Performance analysis—allows a developer to identify and resolve Java bottlenecks and deadlocks, to ensure optimal program performance and scalability.
- Coverage analysis—allows a developer to identify un-executed lines of code during unit testing, to ensure test coverage and program correctness.

JProbe also offers an Eclipse plugin that provides intelligent code performance analysis and problem resolution from within the Eclipse Java IDE.

About This Guide

The *JProbe User Guide* tells you how to configure JProbe to run your application, start your application from JProbe, run JProbe analysis sessions, analyze your results, and communicate those results to others. It also includes information on troubleshooting, strategies for using JProbe, and other general help.

This document is intended for Java developers who want to analyze the memory usage and performance of their code or find out how well their test cases cover their code.

The *JProbe User Guide* is organized as follows:

Chapter 1, Configuring JProbe to Analyze Your Application—provides detailed information about the process of configuring JProbe to run your Java EE or Java SE application.

Chapter 2, Starting Your Application with JProbe Attached—explains how to run your application under JProbe and connect a JProbe Console to the JProbe Analysis Engine. It also presents how to attach a JProbe Console to an application started without JProbe.

Chapter 3, Strategies for Memory Analysis—provides recommendations for how to use the JProbe Memory analysis tool in your development environment.

Chapter 4, Running a Memory Analysis—describes how to run an analysis session, locate and investigate loitering objects, and presents some techniques for solving memory problems.

Chapter 5, Learning More About Memory Options—presents how to fine-tune the Memory analysis tool and how to analyze a heap dump.

Chapter 6, Exploring Memory Analysis Views—provides detailed information about the Memory views used to present memory data.

Chapter 7, Reporting Memory Analysis Results—presents the Memory reports.

Chapter 8, Troubleshooting Memory Analysis Results—presents a few tips for solving some Memory analysis problems and for running sessions efficiently.

Chapter 9, Strategies for Performance Analysis—describes strategies for using the JProbe Performance analysis tool to uncover performance issues in your application data.

Chapter 10, Running a Performance Analysis—describes how to run an analysis session and investigate your application's performance.

Chapter 11, Learning More About Performance Options—presents how to fine-tune the Performance analysis tool and how to analyze a heap dump.

Chapter 12, Exploring Performance Analysis Views—provides detailed information about the Performance views used to present performance data.

Chapter 13, Reporting Performance Analysis Results—presents the Performance reports.

Chapter 14, Troubleshooting Performance Analysis Results—presents a few tips for solving some Performance analysis problems and for running sessions efficiently.

Chapter 15, Strategies for Coverage Analysis—provides recommendations for how to use the JProbe Coverage analysis tool in your development environment.

Chapter 16, Running a Coverage Analysis—presents a basic analysis using default Coverage settings.

Chapter 17, Learning More About Coverage Options—presents how to fine-tune the Coverage analysis tool.

Chapter 18, Exploring Coverage Analysis Views—provides detailed information about the Coverage views used to present coverage data.

Chapter 19, Reporting Coverage Analysis Results—presents the Coverage reports and the report options.

Chapter 20, Troubleshooting Coverage Analysis Results—presents a few tips for solving some Coverage analysis problems and for running sessions efficiently.

Chapter 21, Learning How to Use JProbe Effectively—provides background information on how JProbe works and suggests some things that you can do to use JProbe effectively.

Chapter 22, Refining Data Collection and Presentation—provides detailed information about JProbe filters, triggers, and categories.

Chapter 23, Communicating Results—presents how to create reports, print to PDF, save data to a CSV file, and save charts as images using JProbe. It also presents how you can view the analysis logs and the session history, and how to send a report bundle to Quest Support.

Chapter 24, Managing Snapshots—presents how to manage the snapshots in your project.

Chapter 25, Setting Preferences—presents how to set preferences that affect how JProbe operates and displays data.

Chapter 26, Running Remote Sessions—describes how to run remote JProbe sessions.

Chapter 27, JProbe License Options—provides information about the license server that allows you to run JProbe sessions, and details about the license options.

JProbe Documentation Suite

The JProbe documentation suite is provided in a combination of online help, PDF, HTML, and TXT.

- **Online Help:** You can open the online help by clicking the **Help** icon on the JProbe toolbar.
- **PDF:** The complete JProbe documentation set is available in PDF format on SupportLink. The PDF documentation can also be found in the Documentation folder on the JProbe DVD. The default location of the documentation after an installation is *<jprobe_home>/docs*. Adobe Reader is required.
- **HTML:** *Release Notes* are provided in HTML and TXT format. The default location of this document after an installation is *<jprobe_home>/docs*.

The *Ant Tasks User Manual* is also provided in HTML format. The default location of this document after an installation is *<jprobe_home>/automation/doc*. To open it, navigate to *index.html*.

Core Documentation Set

The core documentation set consists of the following files:

- *JProbe Installation Guide* (PDF)
- *JProbe User Guide* (PDF and online help)
- *JProbe Reference Guide* (PDF)
- *JProbe Plugins for Eclipse Guide* (PDF)
- *JProbe Tutorials* (PDF and online help)
- *JProbe Release Notes* (HTML and TXT)
- *Ant Tasks User Manual* (HTML)

Feedback on the Documentation

We are interested in receiving feedback from you about our documentation. For example, did you notice any errors in the documentation? Were any features undocumented? Do you have any suggestions on how we can improve the documentation? All comments are welcome. Please submit your feedback to the following email address:

am.docfeedback@quest.com

Please do not submit Technical Support related issues to this email address.

Text Conventions

The following table summarizes how text styles are used in this guide:

Convention	Description
`Code`	Monospace text represents code, code objects, and command-line input. This includes: • Java language source code and examples of file contents • Classes, objects, methods, properties, constants, and events • HTML documents, tags, and attributes
`Variables`	Monospace-plus-italic text represents variable code or command-line objects that are replaced by an actual value or parameter.
Interface	Bold text is used for interface options that you select (such as menu items) as well as keyboard commands.
Files, components, and documents	Italic text is used to highlight the following items: • Pathnames, file names, and programs • The names of other documents referenced in this guide

About Quest Software, Inc.

Now more than ever, organizations need to work smart and improve efficiency. Quest Software creates and supports smart systems management products—helping our customers solve everyday IT challenges faster and easier. Visit *www.quest.com* for more information.

Contacting Quest Software

Email	*info@quest.com*
Mail	Quest Software, Inc. World Headquarters 5 Polaris Way Aliso Viejo, CA 92656 USA
Web site	*www.quest.com*

Refer to our web site for regional and international office information.

Contacting Quest Support

Quest Support is available to customers who have a trial version of a Quest product or who have purchased a commercial version and have a valid maintenance contract. Quest Support provides around the clock coverage with SupportLink, our web self-service. Visit SupportLink at: *http://support.quest.com*.

From SupportLink, you can do the following:

- Quickly find thousands of solutions (Knowledgebase articles/documents).
- Download patches and upgrades.
- Seek help from a Support engineer.
- Log and update your case, and check its status.

View the *Global Support Guide* for a detailed explanation of support programs, online services, contact information, and policy and procedures. The guide is available at: *http://support.quest.com/pdfs/Global Support Guide.pdf*.

Quest Communities

Get the latest product information, find helpful resources, and join a discussion with the JProbe Quest team and other community members. Join the JProbe community at: *http://jprobe.inside.quest.com/*.

1

Configuring JProbe to Analyze Your Application

This chapter provides detailed information about the process of configuring JProbe to run your Java EE or Java SE application.

This chapter contains the following sections:

Configuration Overview 24
Configuring a Java EE Application 24
Configuring a Java SE Application 27
Defining a JProbe Configuration 29
Specifying Your Java EE Application Server 31
Specifying Your Java Application 31
Specifying a Java Virtual Machine 36
Specifying Your Code 38
Specifying a JProbe Analysis Type 45
Specifying Advanced JProbe Options 46
Reviewing the Configuration Summary 49
Saving the Configuration 50
Integrating Your Application 51
Saving the Configuration and Integrating 55
Running Your Script 55
Running Your Script and Connecting the Console 58
Managing Configurations 59
What Can I Do if My Application Server Is Not Supported? 64

Configuration Overview

The following sections describe the process of creating a JProbe configuration for your application and attaching JProbe to obtain data for analysis. Before you can do this, you or your system administrator must have completed the following tasks, as outlined in the *JProbe Installation Guide*:

- Installed JProbe on your workstation and specified a valid license.
- For a network installation, also installed JProbe on the server where your application runs and specified a valid license.
- If the JVM that you use to run your application or application server is not supported, installed a supported JVM on the computer where your application runs.

Configuring a Java EE Application

A basic setup takes only a few minutes, though it may take longer if your application is complex or if you want to read more about the available options.

Caution Not all application servers are supported on all platforms. For more information, see the *JProbe Installation Guide*.

Before you begin, you need to have the following information:

- If your application server has a startup script, the path to that script.
- If there is no startup script, the path to the server's *home* and/or *bin* directory.
- The names of packages used in your application.
- The type of analysis you want to run (Memory, Performance, or Coverage).

Depending on your application server, you may be required to provide additional details. For details, see the help provided in the user interface, below each field.

At the end of this process you will have a settings file and a startup script. For some application servers, an additional file may be created to modify (and later restore) one of

the application server's configuration files. If so, the affected file is identified in the online help and also under Running a Java EE Application with JProbe.

Note If you are running JProbe in a heterogeneous, networked environment (for example, UNIX on one computer and Windows on another), configure JProbe on the computer that runs your application server. This ensures that the correct syntax for the operating system is used.

To set up JProbe to analyze a Java EE application:

1. Launch the JProbe Console on the computer where you run your application server.

   Windows: Click **Start** > **Programs** > **JProbe** > **JProbe Console**.

   UNIX: Navigate to *JPROBE_HOME/bin/* and execute `./jpconsole`

2. Click **Configurations** in the Navigation area.

3. Right-click the vendor of the application server hosting your application and click **New Configuration**.

   or

   Select the vendor of the application server hosting your application and click **Configurations** > **New Configuration**.

   Tip Alternatively, you can use the standalone version of the Configuration tool, called *jpconfig*. You can launch *jpconfig* from the **Start > Programs > JProbe > JProbe Config** menu in Windows, or from the *JPROBE_HOME/bin* directory. In a headless environment you can use the command-line version of *jpconfig*. For detailed instructions, see the *JProbe Reference Guide*.

   The **New Configuration Wizard** dialog box appears.

   Tip If your application server is not in the list, you may still be able to use JProbe. For more information, see "What Can I Do if My Application Server Is Not Supported?" on page 64.

4. Select a configuration type, provide a name for this configuration, and click **Next**.

   For more information, see "Defining a JProbe Configuration" on page 29.

5. Enter the configuration data of your application server installation. Select a supported JVM, type any Java options that you use to run your application, and click **Next**.

   For more information, see "Specifying Your Java EE Application Server" on page 31.

6 Specify application filters, enable JProbe to create a category containing the code set to `include`, and click **Next**.

For more information, see “Specifying Your Code” on page 38.

7 Select the type of analysis you want to run.

For more information, see “Specifying a JProbe Analysis Type” on page 45.

In this screen, you have the following options:

- Continue the configuration process, by specifying advanced JProbe settings. Click **Next** and continue on step 8.
- Save the configuration settings and close the configuration wizard. Click **Save & Close**. For more information, see “Saving the Configuration” on page 50.
- Save the configuration settings and integrate your application to run with JProbe. Click **Save & Integrate**. For more information, see “Saving the Configuration and Integrating” on page 55.

8 **Optional**: Specify the JProbe advanced options.

For more information, see “Specifying Advanced JProbe Options” on page 46.

In this screen, you have the following options:

- Continue the configuration process, by reviewing the settings. Click **Next** and continue on step 9.
- Save the configuration settings and close the configuration wizard. Click **Save & Close**. For more information, see “Saving the Configuration” on page 50.
- Save the configuration settings and integrate your application to run with JProbe. Click **Save & Integrate**. For more information, see “Saving the Configuration and Integrating” on page 55.

9 **Optional**: Review the settings in the configuration summary screen.

For more information, see “Reviewing the Configuration Summary” on page 49.

The configuration is now complete. In this screen, you have the following options:

- Edit any of the configuration settings, if necessary. Click **Edit** beside the setting that needs further updates. The wizard opens the screen corresponding to the selected setting, then guides you through the appropriate configuration steps.

  Caution You may need to define some additional settings in the subsequent wizard screens (for example, filters and elements).

- Save the configuration settings and close the configuration wizard. Click **Save & Close**. For more information, see "Saving the Configuration" on page 50.
- Save the configuration settings and integrate your application to run with JProbe. Click **Save & Integrate**. For more information, see "Saving the Configuration and Integrating" on page 55.

Configuring a Java SE Application

A basic setup takes only a few minutes, though it may take longer if your application is complex or if you want to read more about the available options.

Note To integrate an unsupported application server, follow the procedure in section "Creating a Configuration for an Unsupported Application Server" on page 65.

Before you begin, you should have the following information:

- The path to the main class of your application.
- The path to the supported JVM you want to use.
- Any arguments you use on the command line.
- The path to all class libraries required by your application, if not specified in your default class path.
- The names of packages used in your application.
- The type of analysis you want to run (Performance, Memory, or Coverage).

At the end of this process you will have a settings file and a startup script.

Note If you are running JProbe in a heterogeneous, networked environment (for example, UNIX on one computer and Windows on another), configure JProbe on the computer that runs your application. This ensures that the correct syntax for the operating system is used.

To set up JProbe to analyze a Java SE application:

1. Launch the JProbe Console on the computer where you run your application.

   Windows: Click **Start > Programs > JProbe > JProbe Console**.

   UNIX: Navigate to *JPROBE_HOME/bin/* and execute `./jpconsole`

2. Click **Configurations** in the Navigation area.

3 Right-click **Java Application** and click **New Configuration**.

or

Select **Java Application** and click **Configurations > New Configuration**.

Tip Alternatively, you can use the standalone version of the Configuration tool, called *jpconfig*. You can launch *jpconfig* from the **Start > Programs > JProbe > JProbe Config** menu in Windows, or from the *JPROBE_HOME/bin* directory. In a headless environment you can use the command-line version of *jpconfig*. For detailed instructions, see the *JProbe Reference Guide*.

The **New Configuration Wizard** dialog box appears.

Tip If your application server is not in the list, you may still be able to use JProbe. For more information, see "What Can I Do if My Application Server Is Not Supported?" on page 64.

4 Provide a name for this configuration, and click **Next**.

For more information, see "Defining a JProbe Configuration" on page 29.

5 Specify the main class of your application, working directory, the class path, and any application arguments, and click **Next**.

For more information, see "Specifying Your Java Application" on page 31.

6 Specify a supported JVM, plus any Java options you normally use, and click **Next**.

For more information, see "Specifying a Java Virtual Machine" on page 36.

7 Specify application filters, enable JProbe to create a category containing the code set to `include`, and click **Next**.

For more information, see "Specifying Your Code" on page 38.

8 Select the type of analysis you want to run.

For more information, see "Specifying a JProbe Analysis Type" on page 45.

In this screen, you have the following options:

- Continue the configuration process, by specifying advanced JProbe settings. Click **Next** and continue on step 9.
- Save the configuration settings and close the configuration wizard. Click **Save & Close**. For more information, see "Saving the Configuration" on page 50.
- Save the configuration settings and integrate your application to run with JProbe. Click **Save & Integrate**. For more information, see "Saving the Configuration and Integrating" on page 55.

9 **Optional**: Specify the JProbe advanced options.

For more information, see “Specifying Advanced JProbe Options” on page 46.

In this screen, you have the following options:

- Continue the configuration process, by reviewing the settings. Click **Next**. Continue on step 10.
- Save the configuration settings and close the configuration wizard. Click **Save & Close**. For more information, see “Saving the Configuration” on page 50.
- Save the configuration settings and integrate your application to run with JProbe. Click **Save & Integrate**. For more information, see “Saving the Configuration and Integrating” on page 55.

10 **Optional**: Review the settings in the configuration summary screen.

For more information, see “Reviewing the Configuration Summary” on page 49.

The configuration is now complete. In this screen, you have the following options:

- Edit any of the configuration settings, if necessary. Click **Edit** beside the setting that needs further updates. The wizard opens the screen corresponding to the selected setting, then guides you through the appropriate configuration steps.

  Caution You may need to define some additional settings in the subsequent wizard screens (for example, filters and elements).

- Save the configuration settings and close the configuration wizard. Click **Save & Close**. For more information, see “Saving the Configuration” on page 50.
- Save the configuration settings and integrate your application to run with JProbe. Click **Save & Integrate**. For more information, see “Saving the Configuration and Integrating” on page 55.

Defining a JProbe Configuration

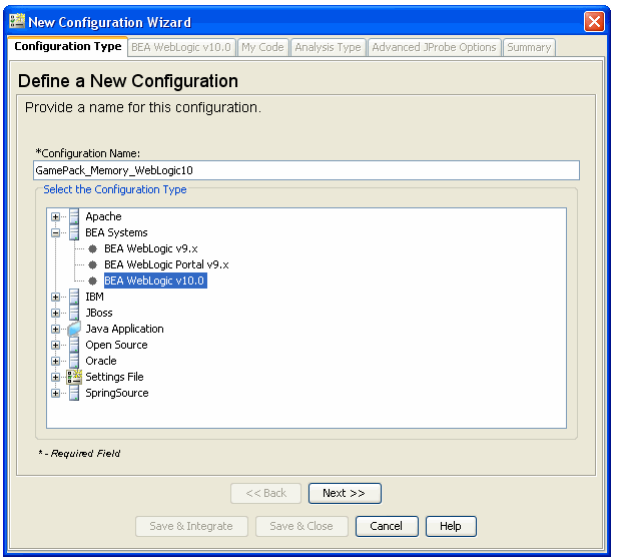
You can think of a configuration as containing everything necessary to run your application under JProbe. A JProbe configuration contains the following information:

- For Java EE applications, the application server you use and details about how it is set up in your environment.
- For Java SE applications, the main class, working directory, class path, application arguments, and JVM you use.
- The code that you want to focus on in the analysis.

- The type of JProbe analysis (Memory, Performance, or Coverage) that you want to run on your application and specific options for that analysis type.
- Advanced JProbe options (optional).

When naming a configuration, you may want to include words that make it easy to distinguish among your configurations. You could specify the application name, the application server (Java EE) or the JVM (Java SE) used, and the type of analysis. For example, configuration to analyze the memory usage of the JProbe Leak Example demo running on WebLogic 10 could be named “GamePack_WebLogic10_Memory”.

Choose the specific Java EE application server on which your application runs by selecting its name in the **Select the Configuration Type** tree.



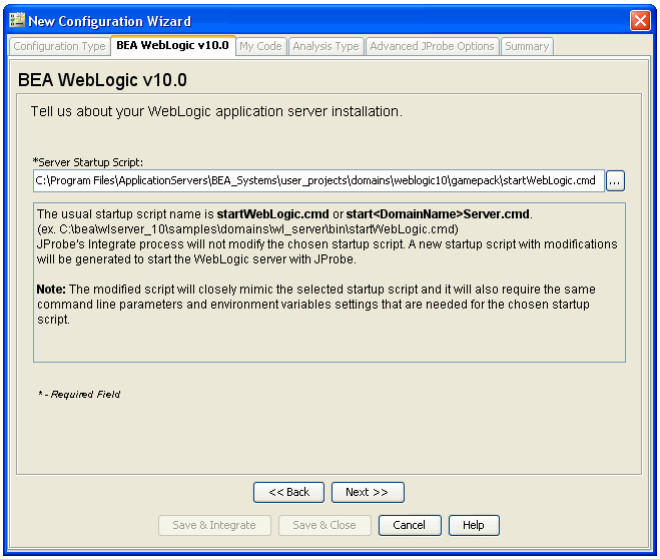
Behind the scenes, a configuration is stored as two files: a settings file and a startup script. For some application servers, JProbe may also copy, edit, and later restore a configuration file for the application server. For more information, see “Integrating Your Application” on page 51.

After you successfully run your first analysis, you may want to run other types of analyses or use different options. Rather than changing your configuration, you could create a new configuration based on the existing configuration. For more information, see “Copying a Configuration” on page 59.

Specifying Your Java EE Application Server

You can specify these options during the wizard on the **<Java EE Application Server Name>** screen, or afterwards in the **<Java EE Application Server Name>** tab.

JProbe needs to know about your installation of the application server. If your application server has a startup script, you specify its location. Otherwise, you specify the path to the server's home directory or possibly a *bin* directory. In some case, you may be required to provide more information, such as a domain name or a process. For details, see the help provided in the user interface below the fields.

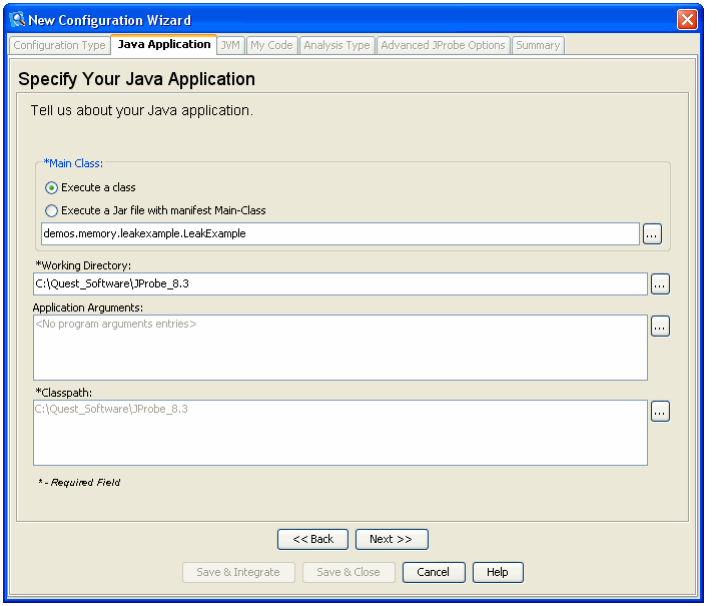

New Configuration Wizard
Configuration Type
BEA WebLogic v10.0
My Code
Analysis Type
Advanced JProbe Options
Summary
BEA WebLogic v10.0
Tell us about your WebLogic application server installation.
*Server Startup Script:
C:\Program Files\ApplicationServers\BEA_Systems\user_projects\domains\weblogic10\gamepack\startWebLogic.cmd
...
The usual startup script name is startWebLogic.cmd or start<DomainName>Server.cmd.
(ex. C:\bea\wlserver_10\samples\domains\wl_server\bin\startWebLogic.cmd)
JProbe's Integrate process will not modify the chosen startup script. A new startup script with modifications will be generated to start the WebLogic server with JProbe.
Note: The modified script will closely mimic the selected startup script and it will also require the same command line parameters and environment variables settings that are needed for the chosen startup script.
* - Required Field
<< Back
Next >>
Save & Integrate
Save & Close
Cancel
Help

For more information, click the **Help** button or see "Runtime Notes for Application Servers" on page 75.

If your application server or version is not listed, you may still be able to use JProbe. For more information, see "What Can I Do if My Application Server Is Not Supported?" on page 64.

Specifying Your Java Application

You can specify these options during the wizard on the **Specify Your Java Application** screen, or afterwards in the **Java Application** tab.



Main Class and Working Directory

You can select the main class or point to a JAR file containing the main class. To be able to select a JAR file, the JAR file must contain a manifest. Manifest files are generated automatically during the archive process or you can specify your own manifest file. For more information, see the Sun Microsystems Web site.

By default, the Main Class dialog box (available from the **Browse** button) lists all the classes. To modify this default behavior, use the **Options** tab in the Main Class dialog box to select the type of files to be filtered out. The following options are available:

- Show only classes with main() method—to include only the main classes.
- Search JAR manifest for main class—to include JAR files that contain a main class.

Both options are by default disabled. They are global options and they are remembered for the next time you open the file chooser.

Important If you have a lot of files in your working directory, the file chooser needs a lot of time to filter out files for the first time. In this case, it is recommended to initially use the default

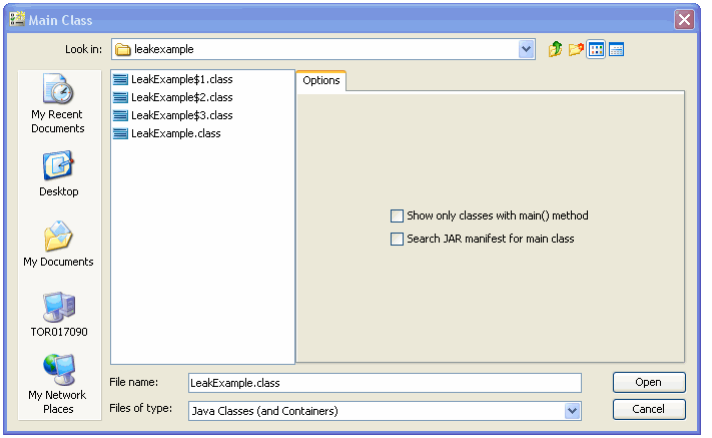
setting (that is, include all the classes), and later filter the list of classes to reduce its complexity.

Later in the wizard process, JProbe creates a default filter in the My Applications Filters table for the package that contains the main class. For more information, see "Specifying Your Code" on page 38.

To specify the main class:

1. Specify whether this main class is a class file (default) or in a JAR file.
2. Click the browse button beside the Main Class field.

   The Main Class dialog box opens.



3. **Optional**: In the **Options** tab, select the check box(es), as necessary.
4. Navigate to and select the class file or JAR file that contains the main method of your program, and click **Open**.

   The Main Class field displays the package and class name. The Working Directory field defaults to the directory containing the package.

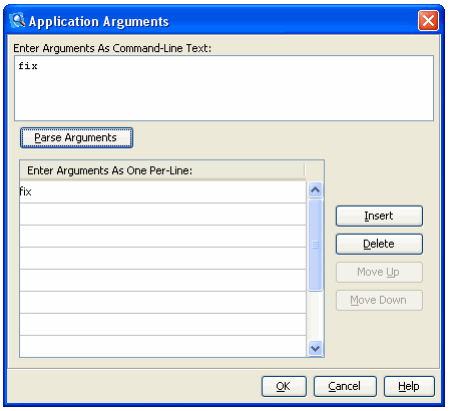
Application Arguments

You can add any application arguments that you would normally pass on the command line. For example, some applications can be set up to accept user input data from a file specified on the command line instead of the GUI or the keyboard.

To add an application argument:

1. Click the browse button beside the Application Arguments text box.
2. Do one of the following:
   - In the text box, enter the arguments as you would on the command line and click **Parse Arguments**.

     The table is populated with a list of arguments. If an argument was parsed incorrectly, double-click the cell to edit the text.
   - In the table, double-click an empty cell and enter an argument.
3. Repeat step 2 for additional arguments.



4. To rearrange or delete arguments from the table, click a row and use the buttons provided.
5. Click **OK**.

   If there is a mismatch between the contents of the text box and the table, JProbe prompts you to select which set of options to use. The default is the table. The arguments are listed in the Application Arguments text box.

Classpath

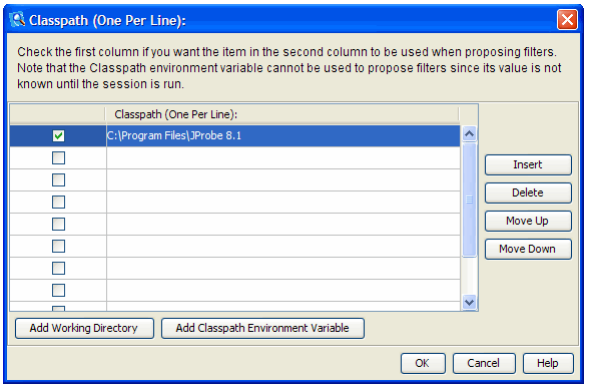
Add all the paths that are required by your application in the Classpath field, as you would if you were using the `-classpath` option on the command line (which is essentially what JProbe does behind the scenes). It is not a good idea to rely on the CLASSPATH environment variable because it can change over time.

If you want JProbe to know about packages in the paths you specify, select the check box beside the path. JProbe can use this information to create default filters in the My Application Filters table for these packages. For more information, see “Specifying Your Code” on page 38.

Tip If you create your settings on Windows and then run it on a UNIX platform, the classpath will contain syntax errors because JProbe would have used semicolons between paths (as required on Windows). To avoid this problem, you can create an environment variable on your UNIX platform that contains the classpath, and then enter the name of the environment variable in the first cell. Alternatively, you can hand-edit the settings file to change semicolons to colons.

To specify the classpath:

1. Click the browse button beside the Classpath text box.
2. To add the working directory to the classpath, click **Add Working Directory**.

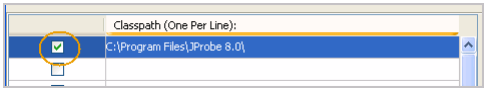

Classpath (One Per Line):
Check the first column if you want the item in the second column to be used when proposing filters. Note that the Classpath environment variable cannot be used to propose filters since its value is not known until the session is run.
Classpath (One Per Line):
C:\Program Files\JProbe 8.1
Insert
Delete
Move Up
Move Down
Add Working Directory
Add Classpath Environment Variable
OK
Cancel
Help

3. To add another path required by your application, click an empty cell and do one of the following:
   - Click the cell browse button, navigate to the path, and click **Open**.
   - Double-click the cell and enter the fully qualified path.
4. To add an environment variable, double-click an empty cell and enter the name of the environment variable using the form `%VAR%` or `${VAR}` (either one works on both Windows and UNIX platforms). Repeat for additional environment variables.

   Note While not advisable, you can choose to use your CLASSPATH environment variable. To do so, click **Add Classpath Environment Variable**.

5 To rearrange or delete paths and environment variables from the table, click a row and use the buttons provided.

6 To have JProbe propose default filters, select the check box beside each path that you want JProbe to search for package names.

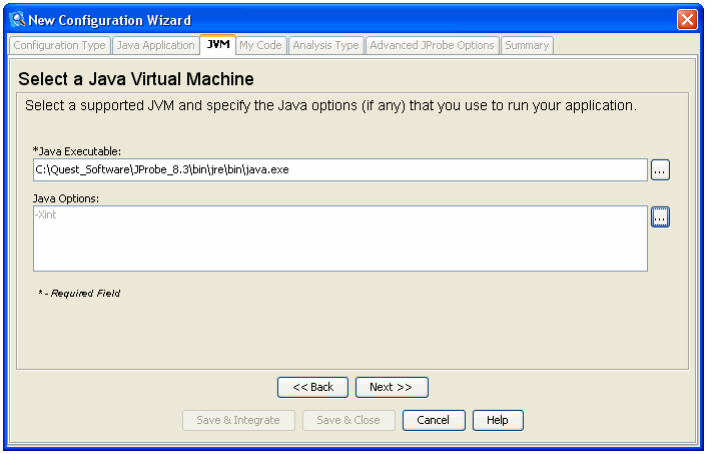


7 Click **OK**.

The classpath is listed in the Classpath text box.

Specifying a Java Virtual Machine

You can specify these options during the wizard on the **Select a Java Virtual Machine** screen, or afterwards in the **JVM** tab.



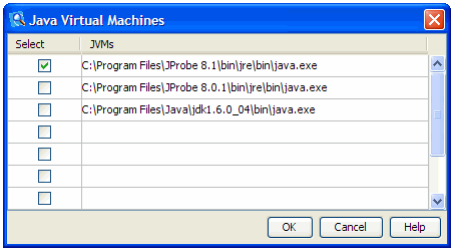
Java Executable

Use a JVM that reflects the JDK environment under which your program was developed and that is also supported by JProbe. JProbe supports the most popular JVMs, however there may be some known restrictions. For a list of supported JVMs and known issues, see the *JProbe Installation Guide*.

To select a JVM:

1. Click the browse button beside the Java Executable field.
2. If your JVM is listed, select its check box.
3. Otherwise, click an empty cell and do one of the following:
   - Click the cell browse button, navigate to the Java executable, and click **Open**.
   - Double-click the cell and enter the fully qualified path to the Java executable.

   The check box for this JVM is automatically selected.



4. Click **OK**.

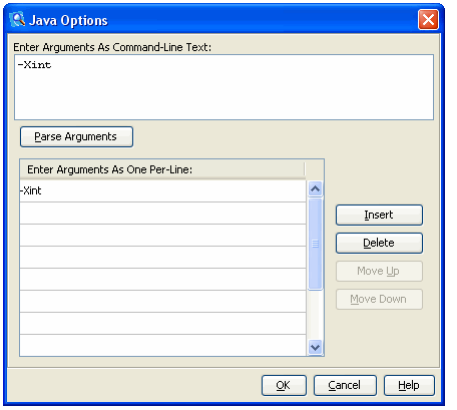
Java Options

You can add Java options as you would on the command line.

To specify Java options:

1. Click the browse button beside the Java Options text box.
2. Do one of the following:
   - Enter the options in the text box as you would on the command line and click **Parse Arguments**.

     The table is populated with a list of options. If an option was parsed incorrectly, you can double-click the cell and edit the text.
   - To enter the options one at a time in the table, double-click an empty cell and enter an option.
3. Repeat for additional options.



4 To rearrange or delete options from the table, click a row and use the buttons provided.

5 Click **OK**.

If there is a mismatch between the contents of the text box and the table, JProbe prompts you to select which set of options to use. The default is the table. The options are listed in the Java Options text box.

Specifying Your Code

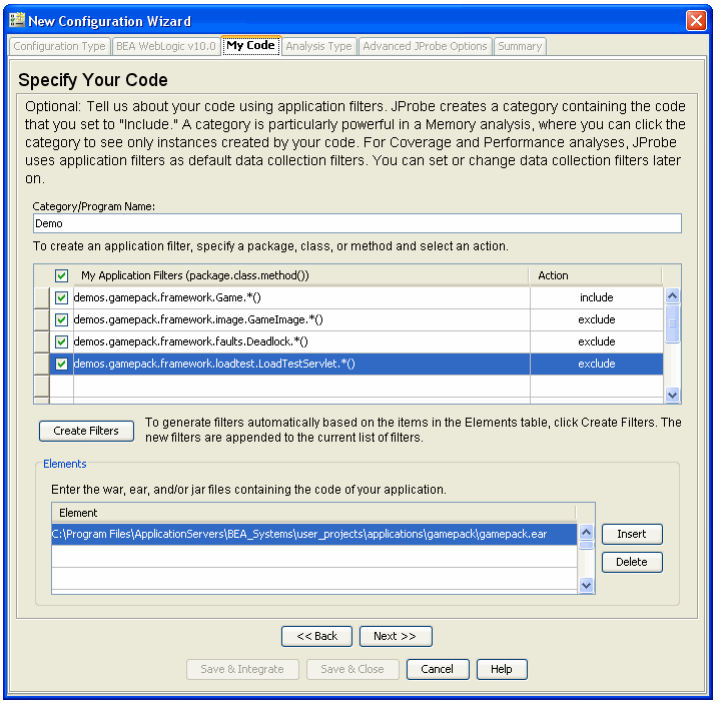
You can specify your code during the wizard on the **Specify Your Code** screen, or afterwards in the **My Code** tab.

The more JProbe knows about your application, the smarter it can be about presenting your data to you. JProbe takes the list of packages, classes, and/or methods that you provide (that is, those set to `include`) and creates a category for your code.

You can define a Category by specifying its name in the **Category/Program Name** text box and its filters in the **My Application Filters** table. When JProbe connects to an application, it looks for a Category with the specified name. If a category with the specified name does not exist, JProbe creates one automatically with the filters you used to specify the code. For more information, see “Creating and Managing Categories” on page 432.

JProbe can also use this information to provide default data collection filters. This means that your analysis collects only the data you are interested in. For more information, see “Refining Data Collection Filters” on page 422.

You specify your custom code using application filters. JProbe may propose some default application filters based on information that you provide elsewhere in the configuration, such as the main class for a Java Application. You can enter other packages (one per row) and/or you can have JProbe create filters for you from elements such as a JAR file, WAR file, or EAR file.


New Configuration Wizard
Configuration Type
BEA WebLogic v10.0
My Code
Analysis Type
Advanced JProbe Options
Summary
Specify Your Code
Optional: Tell us about your code using application filters. JProbe creates a category containing the code that you set to "Include." A category is particularly powerful in a Memory analysis, where you can click the category to see only instances created by your code. For Coverage and Performance analyses, JProbe uses application filters as default data collection filters. You can set or change data collection filters later on.
Category/Program Name:
Demo
To create an application filter, specify a package, class, or method and select an action.
My Application Filters (package.class.method())
Action
demos.gamepack.framework.Game.*()
include
demos.gamepack.framework.image.GameImage.*()
exclude
demos.gamepack.framework.faults.Deadlock.*()
exclude
demos.gamepack.framework.loadtest.LoadTestServlet.*()
exclude
Create Filters
To generate filters automatically based on the items in the Elements table, click Create Filters. The new filters are appended to the current list of filters.
Elements
Enter the war, ear, and/or jar files containing the code of your application.
Element
C:\Program Files\ApplicationServers\BEA_Systems\user_projects\applications\gamepack\gamepack.ear
Insert
Delete
<< Back
Next >>
Save & Integrate
Save & Close
Cancel
Help

My Application Filters Table

If you are configuring a Java Application, you specified a main class for your application. By default, JProbe adds the package containing that class to the My Application Filters table. For more information, see “Main Class and Working Directory” on page 32.

If your Java Application requires other custom packages, or if you are defining a Java EE application, you can add your custom packages to the My Application Filters table. If you want to exclude some of the classes or methods in a package, you can enter additional filters for this code below the package filter and set the action to `Exclude`.

Filters positioned lower in the table override filters on the same code positioned higher in the table.

Note When you exclude from your package the code that you don't need to analyze, JProbe runs faster and records lower overhead during the data collection:

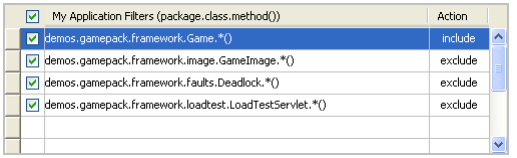
- In the case of a memory analysis, fewer object allocations and releases are recorded during the data recording, which results in fewer instances to analyze.
- In the case of a performance analysis, less method entry/exits are tracked, which results in a smaller call tree/graph.
- In the case of a coverage analysis, less method entry/exits are tracked, which results in fewer classes/packages that are displayed.

To add application filters:

1. Type a category name in the Category/Program name text box.
2. In the My Application Filters table, click an empty cell and do one of the following:
   - Click the cell **Browse** button, navigate to a custom package, and click **Open**.
   - Double-click the cell and enter the package name.

   The package is displayed in the cell and the Action is set to `Include`. If the check box at the beginning of the row is selected, this indicates that the filter is enabled.
3. Repeat for other custom packages.
4. To exclude some code in an included package, click in an empty cell below the package row and do one of the following:
   - To select a subpackage or class, click the cell browse button, navigate to the package or class, and click **Open**. Set the Action to `Exclude`.
   - To select a method, click the cell browse button and navigate to the class containing the method. When you select the class, you can expand its list of methods. Locate and double-click the desired method. Set the Action to `Exclude`.
   - Enter the package, class, or method (no signature) in the cell. For more information, see "Syntax for Filters" on page 41. Set the Action to `Exclude`.
5. Repeat to exclude additional code.
6. **Order is important.** Filters lower in the table override filters defined higher in the table. If you need to rearrange the filters, click and hold the row button

(located at the beginning of the row) for the filter you want to move, and drag it to its new location.



To enable all application filters on the My Application Filters table:

- Select the check box located on the left side of the My Application Filters table title.

  The check boxes corresponding to all application filters on the list are selected automatically. To disable all filters on the list, click the My Application Filters check box again.

To delete a filter from the My Application Filters table:

1. Select the row containing the filter you want to delete from the list.
2. Do one of the following:
   - Right-click the row button (located at the beginning of the row), and click **Delete**.
   - Press **Delete** on the keyboard.

   The selected filter is deleted from the My Application Filters table.

Syntax for Filters

Filters are case-sensitive and blanks are not allowed. You can use an asterisk (*) as a wild card; it matches any character, including the package separator. If you specify the code in a shorthand or partial form, JProbe automatically changes it to canonical form. The change may not be exactly what you want, so be sure to verify it. JProbe does not check to see whether the filter actually matches anything in your code. For example, if you make a typing error in your package name, nothing in your code will match the filter and so the filter is not applied.

The following table contains common syntax and describes how JProbe interprets it.

If you type this:	Canonical form is this:	And filters are applied to:
`*`	`*.*.*()`	All methods in all classes in all packages (including the unnamed package).
`.*`	`.*.*()`	All methods in all classes in the unnamed package only.
`C`	`*.C.*()`	All methods in classes named `C` in any package.
`.C`	`.C.*()`	All methods in the class named `C` in the unnamed package only.
`M()`	`*.*.M()`	Method `M` in all classes in all packages.
`P.String*`	`P.String*.*()`	All methods in any class in the package `P` whose name starts with `String` and all methods in any class in a package or subpackage of `P.String*` (matches both `P.StringArt.draw()` and `P.Stringent.Check.English.spelling()`).
`P.*.C.do*()`	*no change*	Any method whose name starts with `do` in any class named `C` in any subpackage of `P`, except `P` directly (matches `P.SubPkg.C.doGet()`, but does not match `P.C.doNotMatch()`).
`Pre*`	`*.Pre*.*()`	All methods in all classes starting with `Pre` in any package, as well as any class in a subpackage whose name starts with `Pre` (matches both `Pkg.Prepare.m()` and `Pkg.Pretend.C.m()`).

If you type this:	Canonical form is this:	And filters are applied to:
`s*y`	`*.s*y.*()`	All methods in classes whose name ends with `y` and either starts with `s` or belongs to a subpackage that starts with `s` (matches both `java.rmi.server.RMISocketFactory`, `com.quest.say`, and `java.security.cert.X509CRLEntry`).
`foo.bar`	`foo.bar.*()`	All methods in the class `bar` in package `foo`. This is likely not what you wanted. See the next example.
`foo.bar.*`	`foo.bar.*.*()`	All methods in all classes in `foo.bar` or any of its subpackages.
`foo.bar.String.t*e()`	*no change*	Methods whose names start with `t` and end in `e` in `foo.bar.String` (matches both `toUpperCase()` and `toLowerCase()`).

Elements Table

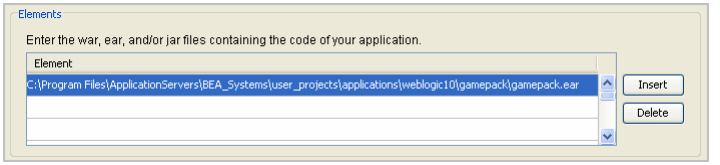
You can specify elements of your application and have JProbe create filters based on the packages and classes that they contain. For this purpose, elements are defined as WAR files, JAR files, EAR files, and any directory containing exploded code.

Tip For Java Applications, when defining your classpath you can select the check box beside any paths that contain exploded code. JProbe automatically displays these paths in the Elements table. For more information, see "Classpath" on page 34.

To create application filters from elements:

1. In the Elements table, click an empty cell and do one of the following:
   - Click the cell **Browse** button, navigate to an element, and click **Open**.
   - Double-click the cell and enter the fully qualified path to the element.
2. Repeat for any other elements.
3. Click **Create Filters**.

The My Application Filters table is updated with a list of filters based on your application elements. By default, all new filters are enabled and appended to the list of filters in the table.



4. To use a filter, select the row check box.
5. Set the filter Action to `Include` (default) or `Exclude` the code.
6. Repeat for any other proposed filters that you want to use.
7. **Order is important.** Filters lower in the table override filters defined higher in the table. If you need to rearrange the filters, click and hold the row button (located at the beginning of the row) for the filter you want to move, and drag it to its new location.

To delete a filter from the My Application Filters table:

1. Select the row containing the filter you want to delete from the list.
2. Do one of the following:
   - Right-click the row button (located at the beginning of the row), and click **Delete**.
   - Press **Delete** on the keyboard.

   The selected filter is deleted from the My Application Filters table.

How JProbe Derives Filters from Elements

Any file or directory can be selected as an element. JProbe examines the elements (files or directories) and proposes filters for all java classes it finds. You get a mixture of package, class, and method filters based on the depth of the package structure and classes existing in the package. Users can select any file or directory and the filter proposal code will traverse all directories inside them.

Element	Code to include as an Application Filter
WAR, JAR, EAR, ZIP, or any other file type	Includes code classes in: • All directories. • The WAR/JAR/EAR/ZIP file.
Directory path	Includes code classes in the directory and its subdirectories.

When JProbe Does Not Propose Filters

JProbe does not propose filters in the following circumstances:

- If the file is not a valid ZIP archive file.
- If the file is not a valid class file.
- If the elements directory does not contain any class file.
- If the elements contain only code that is recognized as System Code or 3rd-Party Frameworks categories (see Creating and Managing Categories).

Specifying a JProbe Analysis Type

You can specify these options during the wizard on the **Select a JProbe Analysis** screen, or afterwards in the **Analysis Type** tab. JProbe offers three analysis types: Memory, Performance, and Coverage. You run one analysis at a time.

The options available for each type of analysis are described in the following sections:

- “Analysis Options for Memory” on page 124
- “Analysis Options for Performance” on page 266
- “Analysis Options for Coverage” on page 368

Why Three Analysis Types?

JProbe targets three separate aspects of code correctness and optimization. Each of the JProbe three analysis engines are optimized to do a very specific job and to do it very well. The different purpose, focus, and timing of these three types of analysis and the importance of accurate results make it the most sensible technical decision to separate the three engines:

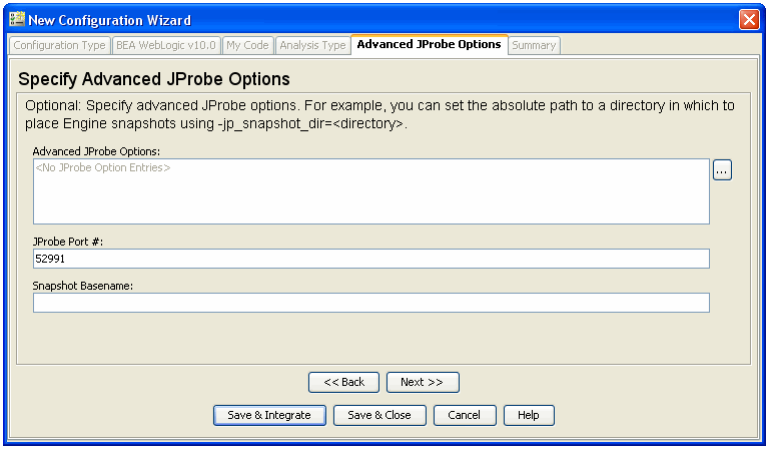
- A Memory analysis requires the highest overhead due to the nature and depth of the object allocation and reference detail being captured. When you are doing Memory analysis, you should not necessarily care about the overhead because you are not measuring the response time, only the memory allocation and reference behavior of your code.
- In a Performance analysis, accurate execution timing is critical, and the Performance tool's low overhead instrumentation does an excellent job of identifying the critical bottleneck code blocks with the least possible overhead. If higher overhead memory detail was being captured at the same time, the Performance results would be much less accurate.
- Coverage analysis is used in conjunction with unit testing and test suite generation. This is often done by a test engineer or QA developer, that is, by someone other than the developer who wrote the code and ran Memory and Performance analyses with JProbe. Coverage analysis is focused on determining which parts of the application have not been tested, so Performance and Memory data is not relevant in this phase of the project.

Which Analysis Should I Do First?

It depends on the types of problems you are seeing in your application. If your application crashes, you probably want to try a Memory analysis first. If your application hangs, run a deadlock detection within your Performance analysis. If your application is slow, our service consultants often recommend a Memory analysis followed by a Performance analysis. The reasoning is that a memory leak will cause the server to crash eventually, so you should address that issue first. After the code is working and running in the context of the entire application, a Performance analysis can help you to fine-tune how your application runs. A Coverage analysis can be done at any time.

Specifying Advanced JProbe Options

You can specify these options during the wizard on the **Specify Advanced JProbe Options** screen, or afterwards in the **Advanced JProbe Options** tab.

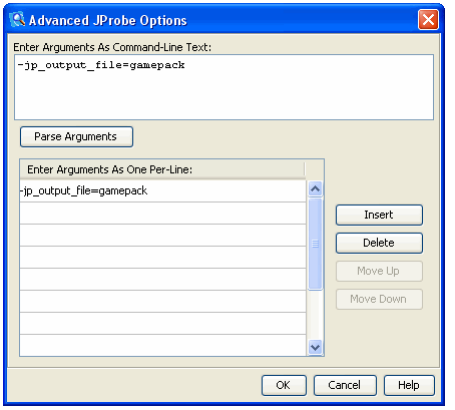


Advanced JProbe Options

This field is for advanced JProbe command line options. For most analyses, you can leave this field blank. Command line options can be useful. For example, you can set an option to change the directory where the JProbe Analysis Engine creates snapshots. For more information, see the list of advanced JProbe options in the “jplauncher” section of the *JProbe Reference Guide*.

To specify JProbe options:

1. Click the browse button beside the Advanced JProbe Options text box.
2. Do one of the following:
   - In the text box, enter the options as you would on the command line and click **Parse Arguments**. The table is populated with a list of options. If an option was parsed incorrectly, double-click the cell and edit it.
   - In the table, double-click an empty cell and enter an option. Repeat for additional options.



3 To re-arrange or delete options from the table, click a row and use the buttons provided.

4 Click **OK**.

If there is a mismatch between the contents of the text box and the table, JProbe prompts you to select a set of options to use. The default is the table. The options are listed in the JProbe Options text box.

JProbe Port

JProbe uses the first available port number starting at 52991. If the port is unavailable, it tries 52992, 52993, and so on. When you run the JProbe-generated startup script for your application, the actual port number used for JProbe is displayed in the JProbe Execution Console window.

If you want to specify a different port number, enter that value in the JProbe Port # field.

Note When you connect the JProbe Console, you may need to update the port number in the **Attach to Running Session** dialog box to match the port number displayed in the JProbe Execution Console window, if the JProbe Analysis Engine is not already displayed with the correct port in the list.

Snapshot Basename

When JProbe takes a snapshot, it automatically assigns the name “snapshot” to the first one and adds an incremental number to subsequent snapshots (snapshot_1, snapshot_2,

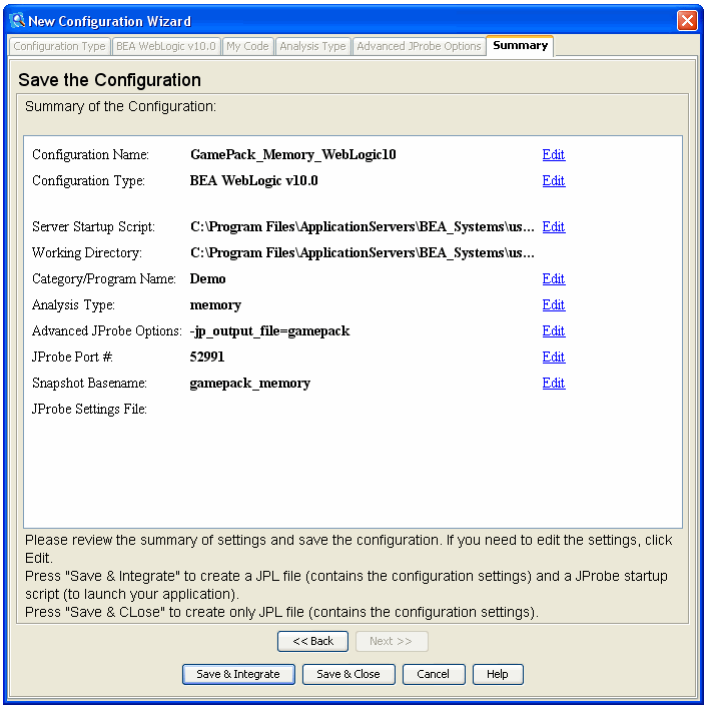
etc.). You can change these default names when you save the snapshots. For more information, see “Saving Snapshots” on page 448.

However, you may want to be able to easily identify snapshots from sessions based on this configuration. If so, you can assign a base name that JProbe will use instead of “snapshot” during a session that is run on the basis of this configuration.

Note The naming convention for snapshots taken using triggers may be different. A base name for a triggered snapshot can be set when the trigger is defined. For more information, see “Setting Triggers” on page 425.

Reviewing the Configuration Summary

Before saving your configuration, review the summary of the settings that you defined.


New Configuration Wizard
Configuration Type
BEA WebLogic v10.0
My Code
Analysis Type
Advanced JProbe Options
Summary
Save the Configuration
Summary of the Configuration:
Configuration Name: GamePack_Memory_WebLogic10 Edit
Configuration Type: BEA WebLogic v10.0 Edit
Server Startup Script: C:\Program Files\ApplicationServers\BEA_Systems\us... Edit
Working Directory: C:\Program Files\ApplicationServers\BEA_Systems\us...
Category/Program Name: Demo Edit
Analysis Type: memory Edit
Advanced JProbe Options: -jp_output_file=gamepack Edit
JProbe Port #: 52991 Edit
Snapshot Basename: gamepack_memory Edit
JProbe Settings File:
Please review the summary of settings and save the configuration. If you need to edit the settings, click Edit.
Press "Save & Integrate" to create a JPL file (contains the configuration settings) and a JProbe startup script (to launch your application).
Press "Save & CLose" to create only JPL file (contains the configuration settings).
<< Back
Next >>
Save & Integrate
Save & Close
Cancel
Help

If a setting is incorrect, you can go back now and change it by clicking the **Edit** link beside the setting. You can also edit your configuration after the wizard ends.

If all settings are correct, you can save the configuration settings, and also integrate your application to run with JProbe.

Tip The **Save & Integrate** option is the recommended choice for first-time users.

Saving the Configuration

When you click **Save & Close**, JProbe saves the configuration settings.

To save the configuration settings:

1. In the **Save JProbe Settings File** dialog box, select a target directory for your settings file.
2. If necessary, rename the settings file. The default file name contains the name of this configuration, the type of analysis, and the word "settings." The file extension is JPL.
3. Click **Save**.

   The new configuration file is saved in the directory of your choice, and the new configuration name is added to the Configurations Navigator. The **<Configuration_File_Name.jpl>** tab appears in the View area. From here you can:

   - Further edit your configuration settings, if necessary. Click **Edit**. The **<Configuration_File_Name.jpl>** tab changes to edit mode. For more information, see "Editing a Configuration" on page 60.
   - Integrate your application to run with JProbe. Click **Integrate**. For more information, see "Integrating Your Application" on page 51.
   - Run your application with JProbe. Click **Run**. For more information, see "Running Your Script" on page 55.
   - Run your application with JProbe and connect the JProbe Console. Click **Run & Connect**. For more information, see "Running Your Script and Connecting the Console" on page 58.

Integrating Your Application

When you select **Integrate**, JProbe creates a startup script. JProbe also verifies that the configuration's settings file is still available and re-saves the settings. For some application servers, a copy of the application server's configuration file may be created and edited.

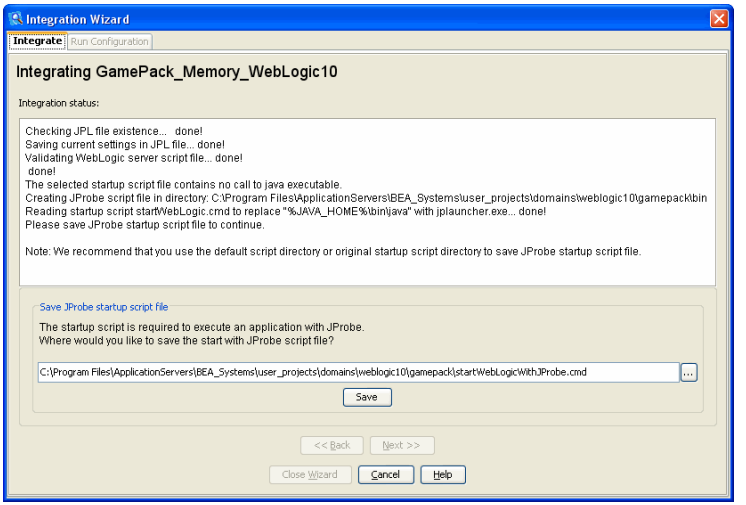
Saving the JProbe Startup Script

To save the JProbe-generated startup script:

1 In the Integration Wizard dialog box, on the **Integrate** tab, review the integration status.

This information is saved to the *Configurations.log* file, which is located in the JProbe temporary files directory (the default on Windows systems is *../Documents and Settings/<user_name>/.jprobe/<JProbe_version>/<computer_name>/*).

Tip You can right-click in the text box to select and copy the text to another location.


Integration Wizard
Integrate
Run Configuration
Integrating GamePack_Memory_WebLogic10
Integration status:
Checking JPL file existence... done!
Saving current settings in JPL file... done!
Validating WebLogic server script file... done!
done!
The selected startup script file contains no call to java executable.
Creating JProbe script file in directory: C:\Program Files\ApplicationServers\BEA_Systems\user_projects\domains\weblogic10\gamepack\bin
Reading startup script startWebLogic.cmd to replace "%JAVA_HOME%\bin\java" with jplauncher.exe... done!
Please save JProbe startup script file to continue.
Note: We recommend that you use the default script directory or original startup script directory to save JProbe startup script file.
Save JProbe startup script file
The startup script is required to execute an application with JProbe.
Where would you like to save the start with JProbe script file?
C:\Program Files\ApplicationServers\BEA_Systems\user_projects\domains\weblogic10\gamepack\startWebLogicWithJProbe.cmd
...
Save
<< Back
Next >>
Close Wizard
Cancel
Help

Review any messages. For example, a configuration for an application server usually includes notes about how to run the JProbe-generated startup script.

Tip Runtime notes are also available from the Help button on the application server's tab (available after the integration is completed) or under "Runtime Notes for Application Servers" on page 75.

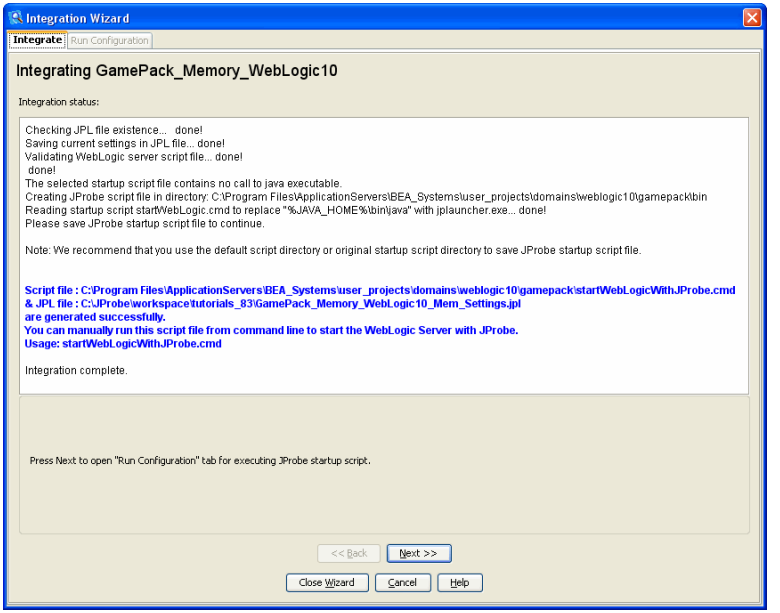
2 If warnings are displayed, you may need to fix some configuration settings. Click **Cancel** and edit the configuration file. For more information, see "Editing a Configuration" on page 60.

3 If no warnings or errors are displayed, specify a location and file name for the startup script.

Caution It is recommended to save the startup script in the default location. Running the startup script from a different directory than the default one, may generate errors for certain types of application servers (for example, JBoss 3.x).

4 Click **Save**.

The script file is saved in the specified location. You are notified when the integration is complete.


Integration Wizard
Integrate
Run Configuration
Integrating GamePack_Memory_WebLogic10
Integration status:
Checking JPL file existence... done!
Saving current settings in JPL file... done!
Validating WebLogic server script file... done!
done!
The selected startup script file contains no call to java executable.
Creating JProbe script file in directory: C:\Program Files\ApplicationServers\BEA_Systems\user_projects\domains\weblogic10\gamepack\bin
Reading startup script startWebLogic.cmd to replace "%JAVA_HOME%\bin\java" with jplauncher.exe... done!
Please save JProbe startup script file to continue.
Note: We recommend that you use the default script directory or original startup script directory to save JProbe startup script file.
Script file : C:\Program Files\ApplicationServers\BEA_Systems\user_projects\domains\weblogic10\gamepack\startWebLogicWithJProbe.cmd
& JPL file : C:\JProbe\workspace\tutorials_83\GamePack_Memory_WebLogic10_Mem_Settings.jpl
are generated successfully.
You can manually run this script file from command line to start the WebLogic Server with JProbe.
Usage: startWebLogicWithJProbe.cmd
Integration complete.
Press Next to open "Run Configuration" tab for executing JProbe startup script.
<< Back
Next >>
Close Wizard
Cancel
Help

5 In this screen, you have the following options:

- Close the integration wizard. Click **Close Wizard**.

The **Integration Wizard** dialog box closes.

- Execute the JProbe startup script. Click **Next**.

  The **Run Configuration** tab is displayed. For more information, see "Executing the JProbe Startup Script" on page 53.

Executing the JProbe Startup Script

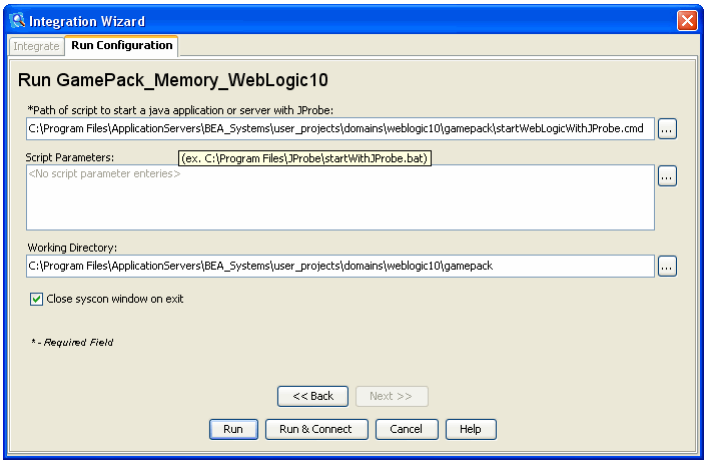
When you are ready to run JProbe on your application, use the JProbe-generated startup script to launch your application. The startup script references the settings file, and JProbe applies these settings when collecting data on your application. For some application servers, the startup script edits (and later restores) one of the application server's configuration files. If so, you are informed during the configuration setup. For more information, look up your application server in "Running a Java EE Application with JProbe" on page 73.

If you are interested in what is happening behind the scenes, you can open the startup script in a file editor. The startup script uses either a standard integration or a hard integration. In a standard integration, you will find that in one or more places where you would normally see a call to `java`, there is instead a call to `jplauncher`. The `jplauncher` is the JProbe tool that does the work of collecting data on your application. For more information on `jplauncher`, see the *JProbe Reference Guide*.

In a hard integration, the startup script passes JProbe settings to Java as a Java option. The settings file that was created from the configuration is not used directly. Rather, JProbe uses the `jplauncher` command `-jp_export_jpl` command to create a JPL file that can be used directly by the JProbe Analysis Engine.

To execute the JProbe-generated startup script:

1. In the Integration Wizard dialog box, on the **Run Configuration** tab, review the startup script settings.



2 Define the following run configuration settings:

- Select the JProbe script to start your application/server. The startup script field is pre-populated with the latest startup script saved for the selected configuration.

  Note If the startup script file needs any environment settings, you should set these environment settings before executing the `jpconfig` or the `jpconsole` application. For example if the script file needs a `JAVA_HOME` value, you should first set this value in your environment, then run `jpconfig` or `jpconsole`, and finally proceed to run the startup script file.

- **Optional**: Type in any script parameters that should be used when launching the script.
- Select your working directory, where the startup script resides.
- **Optional**: Select the **Close syscon window on exit** check box to automatically close the *jpsyscon* window after the termination of the launched script, if the program exits normally. If this check box is cleared, or if the program exits with a non-zero status, you must press **ENTER** to close the *jpsyscon* window.

  Note This option is global and is not saved against each configuration value.

3 In this screen, you have the following options:

- Run your application with JProbe. Click **Run**. The script is launched automatically, and you can start to analyze your application right away. For more information, see “Running Your Script” on page 55.
- Run your application with JProbe and connect the JProbe Console. Click **Run & Connect**. The script is launched automatically, the JProbe Console is

automatically attached to your running application, and you can start to analyze your application right away. For more information, see "Running Your Script and Connecting the Console" on page 58.

Saving the Configuration and Integrating

When you click **Save & Integrate**, JProbe first saves the configuration settings (see "Saving the Configuration" on page 50), then automatically launches the Integration wizard ("Integrating Your Application" on page 51).

Running Your Script

If during the integration stage (see "Executing the JProbe Startup Script" on page 53), you have selected to run the JProbe startup script, the script is launched automatically, and you can start to analyze your application right away.

If you want to specify any additional arguments to the script file or change the working directory, you can use the **Run** button, which opens the Run dialog.

Note Before running the JProbe startup script, you must manually set any environment variable required by your application.

You can run any JProbe script manually, using the JProbe Console or the JProbe Configuration tool. You can also run a JProbe startup script using the command line tools from a command window.

Running Your Script from JProbe Console

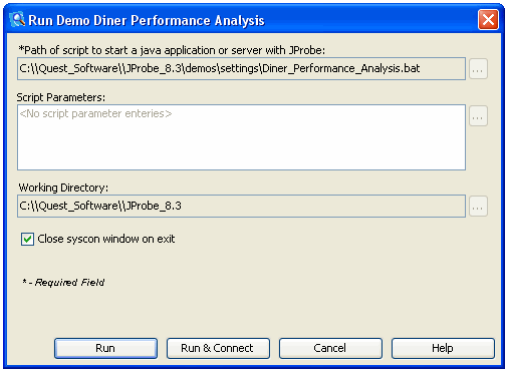
To manually run your JProbe startup script from the JProbe Console:

1. In the JProbe Console, select a configuration from the Configurations Navigator.

   The **<Configuration_File_Name.jpl>** tab appears in the View area.

2. Click **Run**.

   The **Run <Configuration_Name>** dialog box appears.



3 Define the following run configuration settings:

- Select the JProbe script to start your application/server. The startup script field is pre-populated with the latest startup script saved for the selected configuration.

  Note If the startup script file needs any environment settings, you should set these environment settings before executing the `jpconfig` or the `jpconsole` application. For example if the script file needs a `JAVA_HOME` value, you should first set this value in your environment, then run `jpconfig` or `jpconsole`, and finally proceed to run the startup script file.

- **Optional**: Type in any script parameters that should be used when launching the script.
- Select your working directory, where the startup script resides.
- **Optional**: Select the **Close syscon window on exit** check box to automatically close the *jpsyscon* window after the termination of the launched script, if the program exits normally. If this check box is cleared, or if the program exits with a non-zero status, you must press **ENTER** to close the *jpsyscon* window.

  Note This option is global and is not saved against each configuration value.

4 Click **Run**.

The selected application/server launches. You can now connect JProbe to the running application and start analyzing your application.

Running Your Script from JProbe Configuration Tool

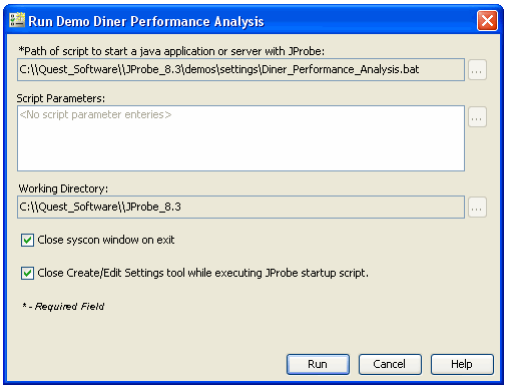
To manually run your JProbe startup script from the JProbe Configuration tool:

1. In the JProbe Configuration tool, select a configuration from the Configurations Navigator.

   The **<Configuration_File_Name.jpl>** tab appears in the View area.

2. Click **Run**.

   The **Run <Configuration_Name>** dialog box appears.



3. Define the following run configuration settings:

   - Select the JProbe script to start your application/server. The startup script field is pre-populated with the latest startup script saved for the selected configuration.

     Note If the startup script file needs any environment settings, you should set these environment settings before executing the `jpconfig` or the `jpconsole` application. For example if the script file needs a `JAVA_HOME` value, you should first set this value in your environment, then run `jpconfig` or `jpconsole`, and finally proceed to run the startup script file.

   - **Optional**: Type in any script parameters that should be used when launching the script.
   - Select your working directory, where the startup script resides.
   - **Optional**: Select the **Close syscon window on exit** check box to automatically close the *jpsyscon* window after the termination of the launched script, if the

program exits normally. If this check box is cleared, or if the program exits with a non-zero status, you must press **ENTER** to close the *jpsyscon* window.

Note This option is global and is not saved against each configuration value.

- **Optional**: Select the **Close Create/Edit Settings tool while executing JProbe startup script** check box to automatically close the Configuration tool window after the termination of the launched script, if the program exits normally.

4 Click **Run**.

The selected application/server launches. You can now open a JProbe Console, connect it to the running application, and start analyzing your application.

Running Your Script and Connecting the Console

If during the integration stage (see "Executing the JProbe Startup Script" on page 53), you have selected to run the JProbe startup script and connect the JProbe Console, the script is launched automatically, the JProbe Console is automatically attached to your running application, and you can start to analyze your application right away.

If you want to specify any additional arguments to the script file or change the working directory, you can use the **Run** button.

Note Before running the JProbe startup script, you must manually set any environment variable required by your application.

You can also run and connect any JProbe script manually, using the JProbe Console.

To manually run your startup script and connect the JProbe Console:

1 In the JProbe Console, select a configuration from the Configurations Navigator.

The **<Configuration_File_Name.jpl>** tab appears in the View area.

2 Click **Run & Connect**.

The selected application/server launches, and the JProbe Console automatically connects to your application. You can start analyzing your application using the Runtime view.

Managing Configurations

This section explains how to copy, edit, and delete configurations, as well as how to manage settings files and review errors. It also provides a list of pre-defined demo configurations installed with JProbe.

Pre-Defined Demo Configurations

When you install JProbe, the following pre-defined configurations are installed and appear in the Configurations Navigator, under:

- Java Applications:
  - Demo Adventure Coverage Analysis—allows you to run a Coverage analysis using the Adventure demo.
  - Demo Diner Performance Analysis—allows you to run a Performance analysis using the Diner demo.
  - Demo Network Memory Analysis—allows you to run a Memory analysis using the Network demo.
- Apache:
  - Demo Gamepack Coverage Analysis—allows you to run a Coverage analysis using the Gamepack demo.
  - Demo Gamepack Memory Analysis—allows you to run a Memory analysis using the Gamepack demo.
  - Demo Gamepack Performance Analysis—allows you to run a Performance analysis using the Gamepack demo.

The demo analysis settings file and the JProbe startup script files are by default loaded from the *<JPROBE_HOME>\demos\settings* directory.

These pre-defined configurations cannot be modified or re-integrated (if you click **Edit**, **Delete**, or **Integrate**, the **Configuration Warning** dialog box opens, notifying you that the operation is not allowed). You can create similar configurations by copying the configuration and modifying it as necessary.

Copying a Configuration

You will likely have more than one configuration. It may be easier to start from a copy of a similar configuration than to begin from scratch. When you copy a configuration,

all the details are preserved. You can then make edits and save the configuration under a new name.

To copy a configuration:

1. Select the configuration in the Configurations Navigator.
2. Right-click the configuration name and select one of the following options:
   - **Copy settings for Memory Analysis**
   - **Copy settings for Performance Analysis**
   - **Copy settings for Coverage Analysis**

   A copy of the original configuration appears in the Configurations Navigator and the configuration settings are displayed in the **Configuration Details** tab.
3. To edit the new configuration, click **Edit**.

   The **Configuration Details** tab is placed in editing mode.

   a. Type in a name for this new configuration.

   b. Edit the settings on any of the tabs, as necessary.

   c. Click **Save**.
4. To generate a startup script, click **Integrate**.

   a. Type in a location and file name for the settings file, and click **Save**.

   b. Click **Continue**.

   c. Type in a location and file name for the startup script, and click **Save**.

   d. Click **Close Wizard**.

Editing a Configuration

When you edit a configuration, the settings are saved to the existing settings file. In most cases, you do not need to reintegrate your configuration.

To edit a configuration:

1. Select the configuration in the Configurations Navigator.

2 In the **<Settings_File_Name>** tab, click **Edit**.

or

Right-click the configuration name and select **Edit**.

or

Click **Configurations > Edit <Configuration_Name>** on the menu bar.

The **<Settings_File_Name>** tab is placed in editing mode.

3 Edit the settings on any of the tabs as necessary.

4 Click **Save**.

Deleting a Configuration

To avoid confusion, you should consider deleting configurations that you no longer use.

To delete a configuration:

1 Select the configuration in the Configurations Navigator.

2 In the **<Settings_File_Name>** tab, click **Delete**.

or

Right-click the configuration name and select **Delete**.

or

Click **Configurations > Delete <Configuration_Name>** on the menu bar.

3 Click **Yes** in the **Delete Confirmation** dialog box.

The configuration is deleted from the Configurations Navigator.

Creating a Configuration from a Settings File

You can create a configuration from a settings file. For example, a colleague may give you a settings file to use as a starting point for your own analysis. During the load process, if the settings file contains invalid options or if there are naming conflicts, JProbe lists the issues in a message box. JProbe then attempts to create a configuration using the valid values in the settings file.

To create a configuration from a settings file:

1 Right-click in the Configurations Navigator and select **Add Settings File**.

The **Load JProbe Settings File** dialog box appears.

2 Navigate to and select the settings file, and click **Open**.

JProbe inspects the content of the selected JPL file. The **Settings File Configuration Wizard** appears.

If the tool recognizes the configuration type, the **Save updated settings to remove errors & warnings** screen appears. Continue with step 4.

If the tool is unable to recognize the configuration type or if the configuration type is missing from the settings file, the **Select configuration type or application server** screen appears.

3 Select the appropriate configuration type from the list and click **Next**.

4 In the **Save updated settings to remove errors & warnings** screen:

a Review the list of errors and warnings and attempt to resolve as many issues as possible before running an analysis using the configuration. Some common warnings you may encounter include:

- Missing JPL file (for example, the JPL file is deleted).
- Missing value for required field.
- Filter format error.
- Trigger format error.
- Invalid or deprecated JProbe options in a JPL file.
- Invalid path.
- Naming conflicts.
- Other exceptions.

Note Any deprecated options are ignored when running the configuration file with a newer JProbe version.

b Review the recommendations presented in the Details section.

c Select one of the following options for saving the updated settings file:

- **Update values in the selected setting file**—to update the values in the selected JPL settings file, and remove any conversion warnings.
- **Don't update values in the selected settings file**—to load the settings from the selected JPL file into a new configuration file. The original JPL settings file remains unchanged. Later on, you can save the updated settings as a new settings file (optional).

- **Save updated values into a settings file name**—to load the settings from the selected JPL file, change any required options, and save the updated settings as a new settings file. The original JPL settings file remains unchanged.

d Click **Save & Close**.

The new configuration is added in the Configurations Navigator.

5 Select the new configuration and click **Edit**.

a Provide a name for this new configuration.

b Review all the settings and edit them as necessary.

c Click **Save**.

6 To generate a startup script, click **Integrate**.

a Type in a location and file name for the settings file, and click **Save**.

b Click **Continue**.

c Type in a location and file name for the startup script, and click **Save**.

d Click **Close Wizard**.

Renaming a Settings File

After you create a configuration, you may want to rename the settings file. You can do this using the **Save As** button. When you save the settings file with a new name, the configuration is updated to reference the new settings file. The startup script is *not* updated. To update the startup script, you need to reintegrate the configuration.

Caution If you forget to integrate, you may end up running a session using an old settings file. If you are renaming a settings file so that you can keep your old settings file and then change your analysis options, the preferred way do this is to copy your configuration. For more information, see "Copying a Configuration" on page 59.

To save settings to a settings file:

1 Select the configuration in the Configurations Navigator.

2 In the **<Settings_File_Name>** tab, click **Save As**.

or

Right-click the configuration name and select **Save As**.

The **Save JProbe Settings File** dialog box appears.

3 Type in a directory and file name for the new settings file and click **Save**.

4 To update your startup script to call the new settings file, click **Integrate**.

5 Follow the prompts to save the files for this configuration.

What Can I Do if My Application Server Is Not Supported?

If your application server is not currently in the list of supported application servers, you may still be able to run JProbe with it.

Caution Unsupported application servers are untested. JProbe may encounter unforeseen issues while collecting data on your application.

You have three options for configuring an unsupported application server. The following table lists your options and states the advantages and disadvantages of each approach.

Option	Advantage/Disadvantage
Creating a Configuration for an Unsupported Application Server	You can create a configuration right now using the **Java Application** option. The disadvantage is that you may miss something and the configuration does not work. The JProbe Community Website (*http://jprobe.inside.quest.com/*) also contains step-by-step instructions for integrating Java application servers like *Jetty HTTP Server* and *Orion Application Server.*
Requesting a Module from Quest Support	You receive a module that you add to your JProbe installation directory. In the Create/Edit Settings dialog box, your application server and/or version is added to the list of available application servers. The disadvantage is that this service is not guaranteed.

Option	Advantage/Disadvantage
Modifying an Application Server Startup Script	You can copy and edit your startup script yourself. This is probably the easiest way and is most likely to be correct. The disadvantage is that you cannot access this configuration from the Create/Edit Settings dialog box.

Creating a Configuration for an Unsupported Application Server

You can use the Java Application option to create a configuration for an unsupported application server.

Before you begin, you need to have the following information:

- The path to the main class of your application server.
- The path to the supported JVM you want to use.
- Any arguments you use on the command line.
- The names of packages used in your application.
- The type of analysis you want to run (Performance, Memory, or Coverage).

At the end of this process you will have a settings file and a startup script.

To create a configuration for an unsupported application server:

1. Verify that you can run your application server with a JProbe-supported JVM. For more information, see the list of supported platforms in the *JProbe Installation Guide*.
1. Launch the JProbe Console on the computer where you run your application server.

   Windows: Click **Start > Programs > JProbe > JProbe Console**.

   UNIX: Navigate to *JPROBE_HOME/bin/* and execute `./jpconsole`
2. Click **Configurations** in the Navigation area.

3 Right-click **Java Application** and click **New Configuration**.

or

Select **Java Application** and click **Configurations > New Configuration**.

Tip Alternatively, you can use the standalone version of the Configuration tool, called *jpconfig*. You can launch *jpconfig* from the **Start > Programs > JProbe > JProbe Config** menu in Windows, or from the *JPROBE_HOME/bin* directory. In a headless environment you can use the command-line version of *jpconfig*. For detailed instructions, see the *JProbe Reference Guide*.

The **New Configuration Wizard** dialog box appears.

Tip If your application server is not in the list, you may still be able to use JProbe. For more information, see "What Can I Do if My Application Server Is Not Supported?" on page 64.

4 Type in a name for this configuration, and click **Next**.

For more information, see "Defining a JProbe Configuration" on page 29.

5 Specify the JAR file used to start your application server, or the main class of your application server and the paths to the classes that the application server requires, and click **Next**.

Note Specifying the JAR file for an application server is preferred, as it may then use the *manifest.mf* file to specify JAR dependencies, so that you do not have to specify them directly within your JProbe settings file.

For more information, see "Specifying Your Java Application" on page 31.

6 Specify a supported JVM, plus any Java options you normally use, and click **Next**.

For more information, see "Specifying a Java Virtual Machine" on page 36.

7 Specify application filters, enable JProbe to create a category containing the code set to `include`, and click **Next**.

For more information, see "Specifying Your Code" on page 38.

8 Select the type of analysis you want to run.

For more information, see "Specifying a JProbe Analysis Type" on page 45.

In this screen, you have the following options:

- Continue the configuration process, by specifying advanced JProbe settings. Click **Next** and continue on step 9.
- Save the configuration settings and close the configuration wizard. Click **Save & Close**. For more information, see "Saving the Configuration" on page 50.

- Save the configuration settings and integrate your application to run with JProbe. Click **Save & Integrate**. For more information, see "Saving the Configuration and Integrating" on page 55.

9 **Optional**: Specify the JProbe advanced options.

For more information, see "Specifying Advanced JProbe Options" on page 46.

In this screen, you have the following options:

- Continue the configuration process, by reviewing the settings. Click **Next** and continue on step 10.
- Save the configuration settings and close the configuration wizard. Click **Save & Close**. For more information, see "Saving the Configuration" on page 50.
- Save the configuration settings and integrate your application to run with JProbe. Click **Save & Integrate**. For more information, see "Saving the Configuration and Integrating" on page 55.

10 **Optional**: Review the settings in the configuration summary screen.

For more information, see "Reviewing the Configuration Summary" on page 49.

The configuration is now complete. In this screen, you have the following options:

- Edit any of the configuration settings, if necessary. Click **Edit** beside the setting that needs further updates. The wizard opens the screen corresponding to the selected setting, then guides you through the appropriate configuration steps.

  Caution You may need to define again some settings in the subsequent wizard screens (for example, filters and elements).

- Save the configuration settings and close the configuration wizard. Click **Save & Close**. For more information, see "Saving the Configuration" on page 50.
- Save the configuration settings and integrate your application to run with JProbe. Click **Save & Integrate**. For more information, see "Saving the Configuration and Integrating" on page 55.

Requesting a Module from Quest Support

Modules are provided on request as a service to Quest Software customers with valid support contracts. However, Quest Software cannot guarantee that all application servers can be supported or that the module can be ready for you within your timeframe.

If time is short, you may want to try one of the other options to integrate your application server.

To request and install a module for your application server:

1. Log a support case. Visit SupportLink and select **Case Management**. Create a case and request a module for your application server.
2. When you receive the module JAR file, save it in the *JPROBE_HOME/modules* directory.
3. Launch JProbe.

   The new application server is displayed in the list of available servers in the Configurations Navigator.
4. Create a configuration as usual.

Note Additional modules may also be made available from the JProbe Community at *http://jprobe.inside.quest.com/*.

Modifying an Application Server Startup Script

You can also configure JProbe for your application server by modifying your application server's startup script manually. If you feel comfortable making changes to your application server's script, you can make a copy of it and replace the main call to `java` with a call to `jplauncher` instead.

Caution When modifying your application server script, make sure that it is not making an external call to another script that starts the application server. This will cause the configuration to fail.

Before you begin, you need a JProbe settings file.

To create a simple settings (JPL) file with only JProbe options, and no application-specific information:

1. Right-click in the Configurations Navigator and select **New Configuration**.

   The **New Configuration Wizard** dialog box appears.
2. Click **Settings File > JProbe Settings File**.
3. Type in a name for this configuration in the **Configuration Name** text box, and click **Next**.

4 **Optional**: Specify a supported JVM, plus any Java options you normally use, and click **Next**.

For more information, see "Specifying a Java Virtual Machine" on page 36.

5 Specify application filters, enable JProbe to create a category containing the code you are interested in (set to `include`), and click **Next**.

For more information, see "Specifying Your Code" on page 38.

6 Select the type of analysis you want to run.

For more information, see "Specifying a JProbe Analysis Type" on page 45.

In this screen, you have the following options:

- Continue the configuration process, by specifying advanced JProbe settings. Click **Next** and continue on step 7.
- Save the configuration settings and close the configuration wizard. Click **Save & Close**. For more information, see "Saving the Configuration" on page 50.
- Save the configuration settings and integrate your application to run with JProbe. Click **Save & Integrate**. For more information, see "Saving the Configuration and Integrating" on page 55.

7 **Optional**: Specify the JProbe advanced options.

For more information, see "Specifying Advanced JProbe Options" on page 46.

In this screen, you have the following options:

- Continue the configuration process, by reviewing the settings. Click **Next** and continue on step 8.
- Save the configuration settings and close the configuration wizard. Click **Save & Close**. For more information, see "Saving the Configuration" on page 50.
- Save the configuration settings and integrate your application to run with JProbe. Click **Save & Integrate**. For more information, see "Saving the Configuration and Integrating" on page 55.

8 **Optional**: Review the settings in the configuration summary screen.

For more information, see "Reviewing the Configuration Summary" on page 49.

The configuration is now complete.

9 Save the configuration settings and close the configuration wizard. Click **Save & Close**.

10 Type in a directory and file name for the new settings file and click **Save**.

The setting (JPL) file is created. This is the JPL file that you use in the following procedure.

To modify the startup script to run an application under JProbe:

1. Verify that you can run your application server with a JProbe-supported JVM. For more information, see the list of supported platforms in the *JProbe Installation Guide*.
2. Make a copy of the application server startup script.
3. In the copy, find `java` and replace it with `jplauncher -jp_input = <JPL_file> -jp_java=<path_to_original_java_executable>`.

   For example, in WebLogic you would find:

   ```
   %JAVA_HOME%\bin\java -Xms64m -Xmx64m -classpath …
   ```

   and replace it with:

   ```
   %JPROBE_HOME%\bin\jplauncher -jp_input=<JPL_file>
   -jp_java=%JAVA_HOME%\bin\java -Xms64m -Xmx64m -classpath …
   ```

   where `<JPL_file>` is the fully qualified path to a JProbe settings file. Use the quoting conventions required by the application server and/or your operating system. You can choose to specify the `java` executable in the JPL file instead of on the command line. If you do not specify the `java` executable, the `java` executable in the `PATH` environment settings is used. You may also need to set `JPROBE_HOME` or use the fully qualified path instead.
4. Ensure that you have added all jar files to the classpath.
5. Save the modified script.
6. To run a JProbe session, execute the modified script.

2

Starting Your Application with JProbe Attached

This chapter explains how to run your application under JProbe and connect a JProbe Console to the JProbe Analysis Engine. It also presents how to attach a JProbe Console to an application started without JProbe.

This chapter contains the following sections:

Running a Java Application with JProbe 72
Running a Java EE Application with JProbe 73
Runtime Notes for Application Servers 75
What to Do When the Console Does Not Connect 95
Connecting a JProbe Console to an Application Started Without JProbe 96

Running a Java Application with JProbe

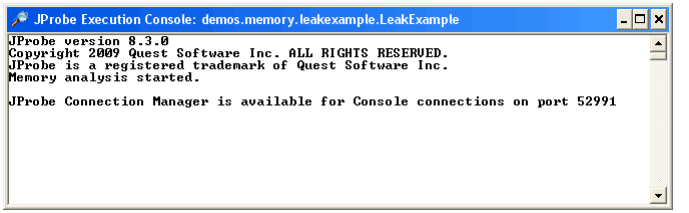
When you integrated your configuration, JProbe created a startup script. To start your application under JProbe, you simply run this startup script. For more information, see “Integrating Your Application” on page 51.

If you are running a remote session, you need to have JProbe installed on both computers. Your license provides for this double installation.

To run your application with JProbe:

1. On the computer where you run your application, run the JProbe-generated startup script from the command line, or from the JProbe Console’s Run dialog. By default, the startup script has the same name as the configuration.

   The JProbe Execution Console window is displayed. It contains the port number that JProbe is using for this session.

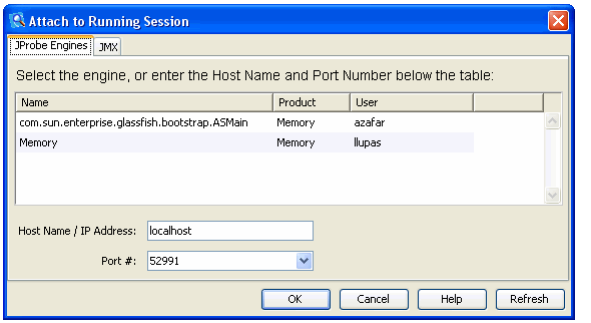

JProbe Execution Console: demos.memory.leakexample.LeakExample
JProbe version 8.3.0
Copyright 2009 Quest Software Inc. ALL RIGHTS RESERVED.
JProbe is a registered trademark of Quest Software Inc.
Memory analysis started.
JProbe Connection Manager is available for Console connections on port 52991

2. To monitor and/or interact with the analysis session:

   a. Launch the JProbe Console on your local computer.

   b. Click **Attach to Session** on the toolbar.

      JProbe searches for currently running engines and JVMs, and displays this list in the **Attach to Running Session** dialog box.



c In the **JProbe Engines** tab, select an engine from the list of currently running engines, or type in the **Host Name/ IP Address** and **Port #** fields the host name and port number of the computer running your application under JProbe.

Tip The computer may be localhost or a remote computer. The port number must be the one specified in the JProbe Execution Console window.

d Click **OK**.

The JProbe Console connects to the JProbe Analysis Engine or to the JVM that is running with the application. The runtime view for the analysis type is displayed.

3 To learn more about monitoring and interacting with sessions, see the "Running an Analysis" topic for the analysis type that you chose.

Running a Java EE Application with JProbe

When you integrated your configuration, JProbe created a startup script for your application server. This is the file that you use to run your application under JProbe. For more information, see "Integrating Your Application" on page 51.

If you are running a remote session, you need to have JProbe installed on both computers. Your license provides for this double installation.

To run your Java EE application with JProbe:

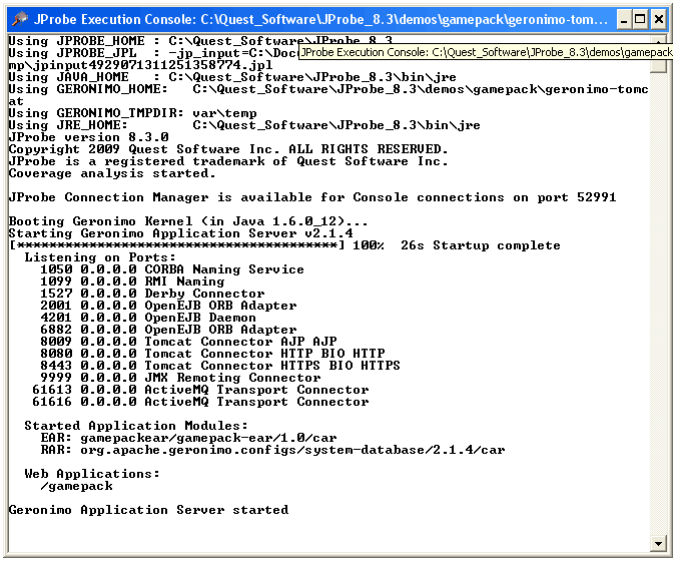
1 **Important:** Review the runtime notes for your application server. See "Runtime Notes for Application Servers" on page 75.

2 If the runtime notes indicate that an application server configuration file will be edited and later restored, make a backup copy of the file.

3 On the computer where you run your application, run the JProbe-generated startup script from the command line, or from the JProbe Console's Run dialog. The syntax is in the runtime notes for your application server.

The JProbe Execution Console window is displayed. It contains the port number that JProbe is using for this session.

Note In some cases, the JProbe Execution Console window does not open, and the application server log files show the JProbe Port number. The application server log file is located in a directory specific to each application server.


JProbe Execution Console: C:\Quest_Software\JProbe_8.3\demos\gamepack\geronimo-tom...
Using JPROBE_HOME : C:\Quest_Software\JProbe_8.3
Using JPROBE_JPL : -jp_input=C:\Doc
JProbe Execution Console: C:\Quest_Software\JProbe_8.3\demos\gamepack
mp\jpinput4929071311251358774.jpl
Using JAVA_HOME : C:\Quest_Software\JProbe_8.3\bin\jre
Using GERONIMO_HOME: C:\Quest_Software\JProbe_8.3\demos\gamepack\geronimo-tomc
at
Using GERONIMO_TMPDIR: var\temp
Using JRE_HOME: C:\Quest_Software\JProbe_8.3\bin\jre
JProbe version 8.3.0
Copyright 2009 Quest Software Inc. ALL RIGHTS RESERVED.
JProbe is a registered trademark of Quest Software Inc.
Coverage analysis started.
JProbe Connection Manager is available for Console connections on port 52991
Booting Geronimo Kernel (in Java 1.6.0_12)...
Starting Geronimo Application Server v2.1.4
[**********] 100% 26s Startup complete
Listening on Ports:
1050 0.0.0.0 CORBA Naming Service
1099 0.0.0.0 RMI Naming
1527 0.0.0.0 Derby Connector
2001 0.0.0.0 OpenEJB ORB Adapter
4201 0.0.0.0 OpenEJB Daemon
6882 0.0.0.0 OpenEJB ORB Adapter
8009 0.0.0.0 Tomcat Connector AJP AJP
8080 0.0.0.0 Tomcat Connector HTTP BIO HTTP
8443 0.0.0.0 Tomcat Connector HTTPS BIO HTTPS
9999 0.0.0.0 JMX Remoting Connector
61613 0.0.0.0 ActiveMQ Transport Connector
61616 0.0.0.0 ActiveMQ Transport Connector
Started Application Modules:
EAR: gamepackear/gamepack-ear/1.0/car
RAR: org.apache.geronimo.configs/system-database/2.1.4/car
Web Applications:
/gamepack
Geronimo Application Server started

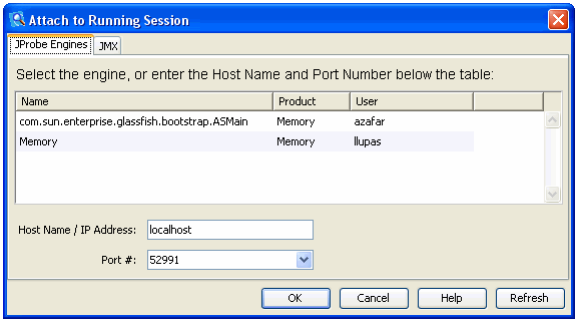
4 Wait for the startup activities to finish.

5 To monitor and/or interact with the analysis session:

a Launch the JProbe Console on your local computer.

b Click **Attach to Session** on the toolbar.

JProbe searches for currently running engines and JVMs, and displays this list in the **Attach to Running Session** dialog box.



c In the **JProbe Engines** tab, select an engine from the list of currently running engines, or type in the **Host Name/ IP Address** and **Port #** fields the host name and port number of the computer running your application under JProbe.

Tip The computer may be localhost or a remote computer. The port number must be the one specified in the JProbe Execution Console window.

d Click **OK**.

The JProbe Console connects to the JProbe Analysis Engine or to the JVM that is running with the application. The runtime view for the analysis type is displayed.

Runtime Notes for Application Servers

The more you know about the changes JProbe makes to its copy of your startup script, the more comfortable you will feel running JProbe on your application. You should take some time to open the JProbe-generated startup script and see what changes have been made. In addition, you need to review the runtime notes for your application server.

Caution Not all application servers are supported on all platforms. For more information, see the "Supported Platforms" chapter of the *JProbe Installation Guide*.

The following sections contain the runtime notes for the application servers.

- "Apache Geronimo v1.0" on page 76
- "Apache Geronimo v1.1" on page 77
- "Apache Geronimo v2.0" on page 77
- "Apache Geronimo v2.1" on page 78
- "Apache Tomcat v5.5" on page 79
- "Apache Tomcat v6.0" on page 80
- "BEA Systems WebLogic v9.x/8.x" on page 81
- "BEA Systems WebLogic Portal v9.x" on page 81
- "BEA Systems WebLogic v10.0" on page 81
- "IBM WebSphere AS Community Edition v2.0.0.1" on page 82
- "IBM WebSphere AS Community Edition v2.1" on page 82
- "IBM WebSphere AS v6.x" on page 83
- "IBM WebSphere Portal Server v6.x" on page 84
- "IBM WebSphere AS v7.0" on page 85

- "JBoss v4.2.x" on page 86
- "JBoss v5.x" on page 87
- "Open Source GlassFish v2 UR2" on page 87
- "Open Source GlassFish v3" on page 88
- "Open Source Java Service Wrapper v3.3" on page 89
- "Oracle AS 10g v10.1.3.0 or below" on page 90
- "Oracle AS 10g v10.1.3.1 or above" on page 91
- "Oracle WebLogic v10.3" on page 92
- "Oracle WebLogic Server 11g R1 (v10.3.1)" on page 92
- "Oracle OC4J 10g v10.0.x" on page 93
- "Oracle OC4J 10g v10.1.2 or below" on page 93
- "Oracle OC4J 10g v10.1.3" on page 93
- "SpringSource dm Server v1.0.2" on page 94
- "SpringSource dm Server v2.0" on page 94

Apache Geronimo v1.0

You run the JProbe-generated startup script in the same way that you run your Geronimo startup script.

Important It is recommended that you use the default script directory (that is, the script directory shown in the startup script field during the integration process) or use the original startup script directory to save the JProbe startup script file. Failing to comply to this recommendation may result in the JProbe startup script not working correctly.

JProbe requires the same environment variables (such as JAVA_HOME) and the same command line arguments.

Usage:

`<configuration_startup_script> [run|start] [geronimo_args]`

where:

- `run` Start Geronimo with JProbe in the current window.

- `start` Start Geronimo with JProbe in a separate window.

Note If you are running Geronimo 1.0, the port number is displayed in GERONIMO_OUT, not in the command window.

Apache Geronimo v1.1

You run the JProbe-generated startup script in the same way that you run your Geronimo startup script.

Important It is recommended that you use the default script directory (that is, the script directory shown in the startup script field during the integration process) or use the original startup script directory to save the JProbe startup script file. Failing to comply to this recommendation may result in the JProbe startup script not working correctly.

JProbe requires the same environment variables (such as JAVA_HOME) and the same command line arguments.

Usage:

`<`*`configuration_startup_script`*`> [run|start] [geronimo_args]`

where:

- `run` Start Geronimo with JProbe in the current window.
- `start` Start Geronimo with JProbe in a separate window.

Note If you are running Geronimo 1.1, the port number is displayed in GERONIMO_OUT, not in the command window.

Apache Geronimo v2.0

You run the JProbe-generated startup script in the same way that you run your Geronimo startup script.

Important It is recommended that you use the default script directory (that is, the script directory shown in the startup script field during the integration process) or use the original startup script directory to save the JProbe startup script file. Failing to comply to this recommendation may result in the JProbe startup script not working correctly.

JProbe requires the same environment variables (such as JAVA_HOME) and the same command line arguments.

Usage:

```
<configuration_startup_script> [run|start] [geronimo_args]
```

where:

- `run` Start Geronimo with JProbe in the current window.
- `start` Start Geronimo with JProbe in a separate window.

Note If you are running Geronimo 2.0, the port number is displayed in GERONIMO_OUT, not in the command window.

Apache Geronimo v2.1

You run the JProbe-generated startup script in the same way that you run your Geronimo startup script.

Important It is recommended that you use the default script directory (that is, the script directory shown in the startup script field during the integration process) or use the original startup script directory to save the JProbe startup script file. Failing to comply to this recommendation may result in the JProbe startup script not working correctly.

JProbe requires the same environment variables (such as JAVA_HOME) and the same command line arguments.

Usage:

```
<configuration_startup_script> [run|start] [geronimo_args]
```

where:

- `run` Start Geronimo with JProbe in the current window.
- `start` Start Geronimo with JProbe in a separate window.

Note If you are running Geronimo 2.1, the port number is displayed in GERONIMO_OUT, not in the command window.

Apache Tomcat v5.5

When creating your JProbe configuration, you indicated whether you start Tomcat as a Windows service (using *tomcat.exe*) or from a startup script. You will run the JProbe-generated startup script differently, depending on that what you indicated.

Important It is recommended that you use the default script directory (that is, the script directory shown in the startup script field during the integration process) or use the original startup script directory to save the JProbe startup script file. Failing to comply to this recommendation may result in the JProbe startup script not working correctly.

Start Tomcat as a Windows service (tomcat.exe)

Usage:

`<configuration_startup_script> [commands]`

where `[commands]` are:

- `run` *Default.* Start Catalina with JProbe in the current window.
- `run -security` Start in the current window with security manager.
- `start` Start Catalina with JProbe in a separate window.
- `start -security` Start in a separate window with security manager.

Start Tomcat from a script

If you start Tomcat from a script, you can run the JProbe-generated startup script in the same way that you run your Tomcat startup script. JProbe requires the same environment variables (such as JAVA_HOME) and the same command line arguments.

Usage:

`<configuration_startup_script> [commands]`

where `[commands]` are:

- `run` Start Catalina with JProbe in the current window.
- `run -security` Start in the current window with security manager.
- `start` Start Catalina with JProbe in a separate window.
- `start -security` Start in a separate window with security manager.

Apache Tomcat v6.0

When creating your JProbe configuration, you indicated whether you start Tomcat as a Windows service (using *tomcat.exe*) or from a startup script. You will run the JProbe-generated startup script differently, depending on what you indicated.

Important It is recommended that you use the default script directory (that is, the script directory shown in the startup script field during the integration process) or use the original startup script directory to save the JProbe startup script file. Failing to comply to this recommendation may result in the JProbe startup script not working correctly.

Start Tomcat as a Windows service (tomcat.exe)

Usage:

`<configuration_startup_script> [commands]`

where `[commands]` are:

- `run` *Default.* Start Catalina with JProbe in the current window.
- `run -security` Start in the current window with security manager.
- `start` Start Catalina with JProbe in a separate window.
- `start -security` Start in a separate window with security manager.

Start Tomcat from a script

If you start Tomcat from a script, you can run the JProbe-generated startup script in the same way that you run your Tomcat startup script. JProbe requires the same environment variables (such as JAVA_HOME) and the same command line arguments.

Usage:

`<configuration_startup_script> [commands]`

where `[commands]` are:

- `run` Start Catalina with JProbe in the current window.
- `run -security` Start in the current window with security manager.
- `start` Start Catalina with JProbe in a separate window.
- `start -security` Start in a separate window with security manager.

BEA Systems WebLogic v9.x/8.x

You run the JProbe-generated startup script in the same way that you run your WebLogic startup script.

Important It is recommended that you use the default script directory (that is, the script directory shown in the startup script field during the integration process) or use the original startup script directory to save the JProbe startup script file. Failing to comply to this recommendation may result in the JProbe startup script not working correctly.

JProbe requires the same environment variables and the same command line arguments.

Usage:

<configuration_startup_script>

BEA Systems WebLogic Portal v9.x

You run the JProbe-generated startup script in the same way that you run your WebLogic startup script.

Important It is recommended that you use the default script directory (that is, the script directory shown in the startup script field during the integration process) or use the original startup script directory to save the JProbe startup script file. Failing to comply to this recommendation may result in the JProbe startup script not working correctly.

JProbe requires the same environment variables and the same command line arguments.

Usage:

<configuration_startup_script>

BEA Systems WebLogic v10.0

You run the JProbe-generated startup script in the same way that you run your WebLogic startup script.

Important It is recommended that you use the default script directory (that is, the script directory shown in the startup script field during the integration process) or use the original startup script directory to save the JProbe startup script file. Failing to comply to this recommendation may result in the JProbe startup script not working correctly.

JProbe requires the same environment variables and the same command line arguments.

Usage:

<configuration_startup_script>

IBM WebSphere AS Community Edition v2.0.0.1

You run the JProbe-generated startup script in the same way that you run your WebSphere startup script.

Important It is recommended that you use the default script directory (that is, the script directory shown in the startup script field during the integration process) or use the original startup script directory to save the JProbe startup script file. Failing to comply to this recommendation may result in the JProbe startup script not working correctly.

JProbe requires the same environment variables (such as JAVA_HOME) and the same command line arguments.

Usage:

<configuration_startup_script> `[run|start] [geronimo_args]`

where:

- `run` Start WebSphere CE Server with JProbe in the current window.
- `start` Start WebSphere CE Server with JProbe in a separate window.

IBM WebSphere AS Community Edition v2.1

You run the JProbe-generated startup script in the same way that you run your WebSphere startup script.

Important It is recommended that you use the default script directory (that is, the script directory shown in the startup script field during the integration process) or use the original startup script directory to save the JProbe startup script file. Failing to comply to this recommendation may result in the JProbe startup script not working correctly.

JProbe requires the same environment variables (such as JAVA_HOME) and the same command line arguments.

Usage:

<configuration_startup_script> `[run|start] [geronimo_args]`

where:

- `run` Start WebSphere CE Server with JProbe in the current window.
- `start` Start WebSphere CE Server with JProbe in a separate window.

IBM WebSphere AS v6.x

When you integrate your configuration, JProbe creates a startup script, resaves your settings file, and also creates a jython script, called *jprobeUpdate.py.* By default, the settings file and the jython script are saved to the same directory; do not separate these files.

When you run the JProbe-generated startup script, you can specify parameters that you usually pass to the WebSphere startup script. Do not, however, specify the server name on the command line because it is set in the configuration. If you do specify a server name, it is ignored. To choose a different server, copy the current configuration, change the server name, and integrate.

Important It is recommended that you use the default script directory (that is, the script directory shown in the startup script field during the integration process) or use the original startup script directory to save the JProbe startup script file. Failing to comply to this recommendation may result in the JProbe startup script not working correctly.

Usage:

`<configuration_startup_script> [options]`

where `[options]` are:

- `-clean` Restores WebSphere to a pre-JProbe state after an abnormal termination.
- `-nowait`
- `-quiet`
- `-logfile <filename>`
- `-replacelog`
- `-trace`
- `-script [<script filename>] [-background]`
- `-timeout <seconds>`
- `-statusport <portnumber>`
- `-recovery`
- `-help`

Before starting the server, the startup script updates the server configuration settings to use `jpjava` executable instead of `java` executable. After starting the server, the script restores the server configuration settings.

If the server does not start successfully, errors are saved to *<WEBSPHERE_HOME>/Appserver/profiles/<PROFILE_NAME>/logs/<SERVER_NAME>/native_stderr.out*.

Caution Do not end this script using **Ctrl+C**. It leaves your application server in an unstable state. Run the JProbe start script with `-clean` argument if an abnormal termination occurs.

WebSphere Profiles

If you want to run JProbe with a WebSphere profile, you need to set the Server Bin Directory field in the configuration to the profile's *bin* directory. You do not need to set the `-profilename <profile>` option.

IBM WebSphere Portal Server v6.x

When you integrate your configuration, JProbe creates a startup script, resaves your settings file, and also creates a jython script, called *jprobeUpdate.py*. By default, the settings file and the jython script are saved to the same directory; do not separate these files.

When you run the JProbe-generated startup script, you can specify parameters that you usually pass to the WebSphere startup script. Do not, however, specify the server name on the command line because it is set in the configuration. If you do specify a server name, it is ignored. To choose a different server, copy the current configuration, change the server name, and integrate.

Important It is recommended that you use the default script directory (that is, the script directory shown in the startup script field during the integration process) or use the original startup script directory to save the JProbe startup script file. Failing to comply to this recommendation may result in the JProbe startup script not working correctly.

Usage:

`<configuration_startup_script> [options]`

where `[options]` are:

- `-clean` Restores WebSphere to a pre-JProbe state after an abnormal termination.
- `-nowait`
- `-quiet`
- `-logfile <filename>`
- `-replacelog`
- `-trace`
- `-script [<script filename>] [-background]`

- `-timeout <seconds>`
- `-statusport <portnumber>`
- `-recovery`
- `-help`

Before starting the server, the startup script updates the server configuration settings to use the `jpjava` executable instead of the `java` executable. After starting the server, the script restores the server configuration settings.

If the server does not start successfully, errors are saved to *<WEBSPHERE_HOME>/Appserver/profiles/<PROFILE_NAME>/logs/<SERVER_NAME>/native_stderr.out*.

Caution Do not end this script using **Ctrl+C**. It leaves your application server in an unstable state. Run the JProbe start script with `-clean` argument if an abnormal termination occurs.

IBM WebSphere AS v7.0

When you integrate your configuration, JProbe creates a startup script, resaves your settings file, and also creates a jython script, called *jprobeUpdate.py*. By default, the settings file and the jython script are saved to the same directory; do not separate these files.

When you run the JProbe-generated startup script, you can specify parameters that you usually pass to the WebSphere startup script. Do not, however, specify the server name on the command line because it is set in the configuration. If you do specify a server name, it is ignored. To choose a different server, copy the current configuration, change the server name, and integrate.

Important It is recommended that you use the default script directory (that is, the script directory shown in the startup script field during the integration process) or use the original startup script directory to save the JProbe startup script file. Failing to comply to this recommendation may result in the JProbe startup script not working correctly.

Usage:

`<configuration_startup_script> [options]`

where `[options]` are:

- `-clean` Restores WebSphere to a pre-JProbe state after an abnormal termination.
- `-nowait`
- `-quiet`
- `-logfile <filename>`

- `-replacelog`
- `-trace`
- `-script [<script filename>] [-background]`
- `-timeout <seconds>`
- `-statusport <portnumber>`
- `-recovery`
- `-help`

Before starting the server, the startup script updates the server configuration settings to use `jpjava` executable instead of `java` executable. After starting the server, the script restores the server configuration settings.

If the server does not start successfully, errors are saved to *<WEBSPHERE_HOME>/Appserver/profiles/<PROFILE_NAME>/logs/<SERVER_NAME>/native_stderr.out*.

Caution Do not end this script using **Ctrl+C**. It leaves your application server in an unstable state. Run the JProbe start script with `-clean` argument if an abnormal termination occurs.

WebSphere Profiles

If you want to run JProbe with a WebSphere profile, you need to set the Server Bin Directory field in the configuration to the profile's *bin* directory. You do not need to set the `-profilename <profile>` option.

JBoss v4.2.x

You run the JProbe-generated startup script in the same way that you run your JBoss startup script.

Important It is recommended that you use the default script directory (that is, the script directory shown in the startup script field during the integration process) or use the original startup script directory to save the JProbe startup script file. Failing to comply to this recommendation may result in the JProbe startup script not working correctly.

JProbe requires the same environment variables (such as JAVA_HOME and JAVA_OPTS) and the same command line arguments:

Usage:

<configuration_startup_script> `[options]`

where `[options]` are:

- `-c, --configuration=<name>` Set the server configuration name.

- `-p, --patchdir=<dir>` Set the patch directory; must be absolute or URL.
- `-n, --netboot=<url>` Boot from net with the given URL as base.

JBoss v5.x

You run the JProbe-generated startup script in the same way that you run your JBoss startup script.

Important It is recommended that you use the default script directory (that is, the script directory shown in the startup script field during the integration process) or use the original startup script directory to save the JProbe startup script file. Failing to comply to this recommendation may result in the JProbe startup script not working correctly.

JProbe requires the same environment variables (such as JAVA_HOME and JAVA_OPTS) and the same command line arguments:

Usage:

`<configuration_startup_script> [options]`

where `[options]` are:

- `-c, --configuration=<name>` Set the server configuration name.
- `-p, --patchdir=<dir>` Set the patch directory; must be absolute or URL.
- `-n, --netboot=<url>` Boot from net with the given URL as base.

Open Source GlassFish v2 UR2

When you integrate your configuration, JProbe creates a startup script, resaves your settings file, and also makes a copy of the GlassFish *domain.xml* file, called *<configuration_startup_script_name>_domain.xml*. In *<configuration_startup_script_name>_domain.xml*, calls to `java` are replaced with calls to `jplauncher`.

Caution You should make a backup copy of the original *domain.xml* file. If you encounter a problem, you can manually restore and load your copy of the *domain.xml* file.

When you run the JProbe-generated startup script, you can specify parameters that you usually pass to the `start-domain` command of *asadmin.bat*. Do not, however, specify

the domain name on the command line because it is set in the configuration. If you do specify a domain name, the application server will not run correctly.

Important It is recommended that you use the default script directory (that is, the script directory shown in the startup script field during the integration process) or use the original startup script directory to save the JProbe startup script file. Failing to comply to this recommendation may result in the JProbe startup script not working correctly.

Usage:

`<configuration_startup_script> [options]`

where `[options]` can be (among others):

- `--user`
- `--passwordfile`

The startup script backs up *domain.xml* and copies *<configuration_startup_script_name>domain.xml* to *domain.xml*. After starting the domain, JProbe restores the *domain.xml* file to its original state and deletes the backup.

Caution Do not end this script using **Ctrl+C**. It leaves your application server in an unstable state. If the server is already in an unstable state, restore the original *domain.xml* from the backup copy.

Open Source GlassFish v3

The GlassFish v3 requires the `com.jprobe.*` filter string to be added to the `org.osgi.framework.bootdelegation` property in the Apache Felix configuration file *config.properties*. The *config.properties* file is located in the *GLASSFISH_HOME\\glassfish\\osgi\\felix\\conf* directory.

You must manually add these filters into the *config.properties* file before integrating GlassFish v3 with JProbe. For example, you can update the property by using the following command:

```
org.osgi.framework.bootdelegation=javax.transaction,
javax.transaction.*, com.jprobe.*
```

Note An error message is displayed when no `com.jprobe.*` filter strings exist in the Apache Felix configuration file.

When you integrate your configuration, JProbe creates a startup script, resaves your settings file, and also makes a copy of the GlassFish *domain.xml* file, called *<configuration_startup_script_name>_domain.xml*. In *<configuration_startup_script_name>_domain.xml*, calls to `java` are replaced with calls to `jplauncher`.

Caution You should make a backup copy of the original *domain.xml* file. If you encounter a problem, you can manually restore and load your copy of the *domain.xml* file.

When you run the JProbe-generated startup script, you can specify parameters that you usually pass to the `start-domain` command of *asadmin.bat*. Do not, however, specify the domain name on the command line, because it is set in the configuration. If you do specify a domain name, the application server will not run correctly.

Important It is recommended that you use the default script directory (that is, the script directory shown in the startup script field during the integration process) or use the original startup script directory to save the JProbe startup script file. Failing to comply to this recommendation may result in the JProbe startup script not working correctly.

Usage:

<configuration_startup_script> `[options]`

where `[options]` can be (among others):

- `--user`
- `--passwordfile`

The startup script backs up *domain.xml* and copies *<configuration_startup_script_name>domain.xml* to *domain.xml*. After starting the domain, JProbe restores the *domain.xml* file to its original state and deletes the backup.

Caution Do not end this script using **Ctrl+C**. It leaves your application server in an unstable state. If the server is already in an unstable state, restore the original *domain.xml* from the backup copy.

Open Source Java Service Wrapper v3.3

The Java Service Wrapper is an application used to install:

- on MS Windows systems—a Java Application or a Java Application Server as a Windows Service.

- on Unix systems—a Java Application as a daemon process.

This module can be used to start different Application Servers and Java Applications configured with the Java Service Wrapper Tanuki Software.

The JProbe configuration module requires two attributes to be set to integrate the Java Service Wrapper with JProbe: the wrapper startup batch file and *wrapper.conf* configuration file. The batch file does not require any modification. It is used only to start the application with JProbe.

The integration process creates a new *wrapper.conf* file with modifications to integrate the Java Application with JProbe. Later, this file is passed as a command line argument to the startup batch file for starting the server with JProbe.

If you want to start the application server on a command line window with JProbe, you must provide the path to the JProbe configuration file as command line argument to the script file.

Example:

- for MS Windows:

  ```
  App.bat <complete path to JProbe's wrapper.conf file>
  ```

- for Unix:

  ```
  App.sh.script <complete path to JProbe's wrapper.conf file>
  ```

Oracle AS 10g v10.1.3.0 or below

When you integrate your configuration, JProbe creates a startup script, saves your settings file, and also makes a copy of the Oracle Application Server's *opmn.xml* file, called <*startup_script_name*>*_opmn.xml*. In <*startup_script_name*>*_opmn.xml*, java-bin data value is replaced with the path to the `jplauncher` executable.

Caution You should make a backup copy of the original *opmn.xml* file. If you encounter a problem, you can manually restore and load your copy of the *opmn.xml* file.

Usage:

<configuration_startup_script>

The startup script backs up *opmn.xml* and copies <*startup_script_name*>*_opmn.xml* to *opmn.xml*. After loading the updated *opmn.xml* and starting the OC4J process, JProbe restores the *opmn.xml* file to its original state and deletes the backup.

Caution Do not end this script using **Ctrl+C**. It leaves your application server in an unstable state.

Important It is recommended that you use the default script directory (that is, the script directory shown in the startup script field during the integration process) or use the original startup script directory to save the JProbe startup script file. Failing to comply to this recommendation may result in the JProbe startup script not working correctly.

Oracle AS 10g v10.1.3.1 or above

When you integrate your configuration, JProbe creates a startup script, saves your settings file, and also makes a copy of the Oracle Application Server's *opmn.xml* file, called <*startup_script_name*>*_opmn.xml*. In <*startup_script_name*>*_opmn.xml*, java-bin data value is replaced with the path to the `jplauncher` executable.

Caution You should make a backup copy of the original *opmn.xml* file. If you encounter a problem, you can manually restore and load your copy of the *opmn.xml* file.

Usage:

<configuration_startup_script>

The startup script backs up *opmn.xml* and copies <*startup_script_name*>*_opmn.xml* to *opmn.xml*. After loading the updated *opmn.xml* and starting the OC4J process, JProbe restores the *opmn.xml* file to its original state and deletes the backup.

Caution Do not end this script using **Ctrl+C**. It leaves your application server in an unstable state.

Important It is recommended that you use the default script directory (that is, the script directory shown in the startup script field during the integration process) or use the original startup script directory to save the JProbe startup script file. Failing to comply to this recommendation may result in the JProbe startup script not working correctly.

Oracle WebLogic v10.3

You run the JProbe-generated startup script in the same way that you run your WebLogic startup script.

Important It is recommended that you use the default script directory (that is, the script directory shown in the startup script field during the integration process) or use the original startup script directory to save the JProbe startup script file. Failing to comply to this recommendation may result in the JProbe startup script not working correctly.

JProbe requires the same environment variables and the same command line arguments.

Usage:

<configuration_startup_script>

Note Windows XP and Windows Vista can be used to develop applications, but are not supported by Oracle as production deployment platforms.

Oracle WebLogic Server 11g R1 (v10.3.1)

You run the JProbe-generated startup script in the same way that you run your WebLogic startup script.

Important It is recommended that you use the default script directory (that is, the script directory shown in the startup script field during the integration process) or use the original startup script directory to save the JProbe startup script file. Failing to comply to this recommendation may result in the JProbe startup script not working correctly.

JProbe requires the same environment variables and the same command line arguments.

Usage:

<configuration_startup_script>

Note Windows XP and Windows Vista can be used to develop applications, but are not supported by Oracle as production deployment platforms.

Oracle OC4J 10g v10.0.x

You run the JProbe-generated startup script using the following syntax:

<configuration_startup_script>

Important It is recommended that you use the default script directory (that is, the script directory shown in the startup script field during the integration process) or use the original startup script directory to save the JProbe startup script file. Failing to comply to this recommendation may result in the JProbe startup script not working correctly.

Oracle OC4J 10g v10.1.2 or below

You run the JProbe-generated startup script using the following syntax:

<configuration_startup_script>

Important It is recommended that you use the default script directory (that is, the script directory shown in the startup script field during the integration process) or use the original startup script directory to save the JProbe startup script file. Failing to comply to this recommendation may result in the JProbe startup script not working correctly.

Oracle OC4J 10g v10.1.3

You run the JProbe-generated startup script in the same way that you run your Oracle OC4J startup script.

Important It is recommended that you use the default script directory (that is, the script directory shown in the startup script field during the integration process) or use the original startup script directory to save the JProbe startup script file. Failing to comply to this recommendation may result in the JProbe startup script not working correctly.

JProbe requires the same environment variables (such as ORACLE_HOME, JAVA_HOME, and OC4J_JVM_ARGS) and the same command line arguments (such as `-start`).

Usage:

<configuration_startup_script>

Caution Do not end this script using **Ctrl+C**. It leaves your application server in an unstable state.

SpringSource dm Server v1.0.2

The SpringSource dm Server requires the filter string `com.jprobe.*` to be added to the `org.osgi.framework.bootdelegation` property in *SERVER_HOME\\lib\\java5-server.profile* and *java6-server.profile* files. You must manually add these filters into the files before integrating the SpringSource dm Server with JProbe.

For example, you can update the `org.osgi.framework.bootdelegation` property as follows:

```
org.osgi.framework.bootdelegation = \
org.w3c.*, org.xml.*,\
sun.*, com.jprobe.*
```

Note An error message is displayed during the integration process when no `com.jprobe.*` filter strings exist in the *java6-server.profile* and *java5-server.profile* files.

You run the JProbe-generated startup script in the same way that you run your startup script (*startup.bat* or *startup.sh*). JProbe requires the same environment variables (such as `JAVA_HOME`) and the same command line arguments.

Important It is recommended that you use the default script directory (that is, the script directory shown in the startup script field during the integration process) or use the original startup script directory to save the JProbe startup script file. Failing to comply to this recommendation may result in the JProbe startup script not working correctly.

SpringSource dm Server v2.0

The SpringSource dm Server requires the filter string `com.jprobe.*` to be added to the `org.osgi.framework.bootdelegation` property in the *SERVER_HOME\\lib\\java6-server.profile* file. You must manually add these filters into the file before integrating the SpringSource dm Server with JProbe.

For example, you can update the `org.osgi.framework.bootdelegation` property as follows:

```
org.osgi.framework.bootdelegation = \
org.w3c.*, org.xml.*,\
sun.*, com.jprobe.*
```

Note An error message is displayed during the integration process when no `com.jprobe.*` filter strings exist in the *java6-server.profile* file.

You run the JProbe-generated startup script in the same way that you run your startup script (*dmk.bat* or *dmk.sh*). JProbe requires the same environment variables (such as `JAVA_HOME`) and the same command line arguments.

Important It is recommended that you use the default script directory (that is, the script directory shown in the startup script field during the integration process) or use the original startup script directory to save the JProbe startup script file. Failing to comply to this recommendation may result in the JProbe startup script not working correctly.

What to Do When the Console Does Not Connect

If the JProbe Console does not connect to an application running under JProbe, you can check to see if any of the following issues apply to your situation.

Port Number Mismatch

The JProbe Console may be listening on the wrong port.

In the JProbe Execution Console window (the window that opened when you ran the JProbe-generated startup script), note the port number that the JProbe Analysis Engine is using for this session.

Click **Attach to Session** on the toolbar and ensure that the same port number is specified in the **Port #** field for the selected JProbe Analysis Engine. Alternatively, check the application server log files, in case you use an application server startup script.

If the JProbe Analysis Engine is on a remote machine, you can use the `jpengines` tool on that machine to find the port number that the Analysis Engine is using and to verify that it is actually running. For details about the `jpengines` tool see the *JProbe Reference Guide*.

Incorrect Host Name

Click **Attach to Session** on the toolbar and ensure that the information specified in the **Host Name/ IP Address** matches the host name of the computer running your application under JProbe.

Application Not Started Under JProbe

Make sure that you launched your application using the JProbe-generated startup script, rather than your usual startup script.

Connecting a JProbe Console to an Application Started Without JProbe

This feature has the following restrictions:

- Requires Sun JVM version 1.6 or later.
- Works only with the JProbe Memory analysis tool.
- The ability to Set Recording Level and the **Heap Data** tab are not available, but you can still request garbage collections and take snapshots.

If you are using a Sun JVM version 1.6 or later, you can attach JProbe to the JVM while your application is running. This may be useful in a pre-production environment where you want to analyze a problem as it is happening. For example, your quality assurance team may be running a set of tests during which they notice that the application has slowed down over time. You can attach JProbe to their JVM, take a snapshot of the heap, detach from the JVM, and then analyze the snapshot using the views available with the JProbe Memory analysis tool.

While JProbe is attached to your JVM, you may notice some impact on your application's performance. When you detach JProbe, your JVM returns to its preconnection state.

Connecting JProbe to a Local Application Started Without JProbe

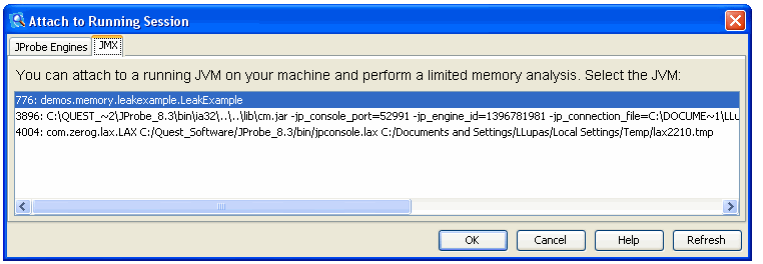
The following instructions apply if the program you want to attach to is running locally, on the same computer you are running the JProbe Console.

To connect a console to a local application started without JProbe:

1. On the JProbe Console, click **Session > Attach to Session**.

   JProbe searches for currently running engines and JVMs, and displays this list in the **Attach to Running Session** dialog box.

2. Click the **JMX** (Java Management Extension) tab.


Attach to Running Session
JProbe Engines
JMX
You can attach to a running JVM on your machine and perform a limited memory analysis. Select the JVM:
776: demos.memory.leakexample.LeakExample
3896: C:\QUEST_~2\JProbe_8.3\bin\ia32\..\..\lib\cm.jar -jp_console_port=52991 -jp_engine_id=1396781981 -jp_connection_file=C:\DOCUME~1\LLu
4004: com.zerog.lax.LAX C:/Quest_Software/JProbe_8.3/bin/jpconsole.lax C:/Documents and Settings/LLupas/Local Settings/Temp/lax2210.tmp
OK
Cancel
Help
Refresh

   In the list of JVMs you see all active JVMs, including the one that is running the JProbe Console (on select platforms).

3. Select the JVM that is running the application and click **OK**.

   JProbe connects to JVM.

   With the exceptions of the Set Recording Level feature and the **Heap Data** tab, which are not available, you can interact with the session in the same way as in any other Memory analysis session, such as requesting garbage collections and taking snapshots.

4. When you have your snapshots, disconnect from the JVM by clicking **Detach From Session**  on the toolbar.

5. Examine the snapshots using the Memory analysis views.

Connecting JProbe to a Remote Application Started Without JProbe

The following instructions apply if the program you want to attach to is running on another computer than the one running the JProbe Console.

To connect a console to a remote application started without JProbe:

1. On the remote machine, from a shell or command window, run the command `jpattach` with no options.

   Note For more information about the `jpattach` command, see the *JProbe Reference Guide*.

   A list of Java applications to which you can attach to is displayed in the command window, as shown in the following example:

   ```
   JProbe version 8.3.0
   Copyright 2009 Quest Software Inc. ALL RIGHTS RESERVED.
   JProbe is a registered trademark of Quest Software Inc.


   Available target JVMs (shown as "vmid: name"):
   13950: demos.memory.leakexample.LeakExample
   13886: demos.memory.network.Network
   ```

2. Select the application to which you want to attach to and run the command `jpattach <vmid>`

   where `<vmid>` is the ID number shown at the left of the application.

   For example, to attach to the *Network* demo included with JProbe, you need to run `jpattach 13886`.

   The command window displays the following output:

   ```
   JProbe version 8.3.0
   Copyright 2009 Quest Software Inc. ALL RIGHTS RESERVED.
   JProbe is a registered trademark of Quest Software Inc.
   Memory analysis attached.

   JProbe Connection Manager is available for Console
   connections on port 52991
   ```

3. On the JProbe Console, click **Session** > **Attach to Session**.

   The **Attach to Running Session** dialog box appears.

4. Do one of the following:

   - If the remote host where you ran `jpattach` and the port appear in the list of found Engines on the **JProbe Engines** tab, while the Product is shown as JMX, then select this entry and click **OK**.
   - If the remote host and port number do not appear in the list of found Engines, enter the name of the remote host where you ran `jpattach` in the Host Name/ IP Address field, and the port number to which `jpattach` printed in the Port # field, and click **OK**.

The JProbe Console connects to the application. With the exceptions of the Set Recording Level feature and the **Heap Data** tab, which are not available, you can interact with the session in the same way as in any other Memory analysis session, such as requesting garbage collections and taking snapshots.

3

Strategies for Memory Analysis

Memory is an analysis tool that helps you to improve memory usage and reduce unnecessary garbage collection in your Java applications. As you run the session, you use the Runtime Summary view to see the overall memory behavior of the program and try to spot classes that may be leaking or are over-allocated. After you capture the data in a memory snapshot, you use the heap investigation tools, starting with the Instances view, to identify leaking instances and find where they were allocated or to find the source of over-allocated objects.

This chapter provides recommendations for how to use the JProbe Memory analysis tool in your development environment.

Tip We recommend analyzing memory usage before measuring performance.

This chapter contains the following sections:

Recording Data.....102
Runtime Summary.....102

Recording Data

After you have started your application and connected JProbe to it, you can set the level of data that you want to record:

- Record Allocations For All Instances
- Record Allocations and Stack Traces For All Instances
- Record Allocations, Stack Traces, and Garbage Data For All Instances
- Record Data as Specified in the Table Below

You can also turn data recording off. These options are explained under "Setting the Recording Level" on page 110.

In Memory, "recording data" means recording information about objects that are allocated. This can be done at varying levels of detail, from the type of data collected (allocation times, allocation traces, or information on garbage collected instances) to the types of instances that are tracked (record only information for instances of packages that you choose).

If you are primarily interested in viewing runtime data, you can run your application and view the data in the Runtime Summary view. While the application is running, you can request a snapshot or a garbage collection and view that data after the analysis session is over.

A snapshot can contain the following types of data:

- The JVM's heap (including all instances, their references, and referrers).
- Any data recorded for individual instances: allocation times, traces, or instances that were garbage collected. The information included depends on the recording options selected.

Note Small amounts of recorded data, especially for large/complex applications, keep the JProbe snapshot size manageable and also make the process of investigation easier.

Runtime Summary

The Runtime Summary is both your view of the Java heap and your control center for running an analysis session. The Runtime Summary helps you to identify loitering objects, excessive garbage collections, and over-allocation of objects, and also acts as a

progress bar. It is the only view available to you until you take a snapshot during or at the end of the session.

When you attach to a session, the Runtime Summary appears automatically. It has three tabs:

- The Memory Pools Tab displays memory usage data.
- The GC Data Tab displays garbage collection data.
- The Heap Data Tab allows you to examine the classes and instance counts in the Java heap of your application, as well as recorded and garbage collected instances tracked thus far.

Toolbar buttons provide access to the snapshot and garbage monitoring controls. For a detailed description of this view, see “Exploring the Memory Runtime Summary View” on page 132.

Memory Pools Tab

The **Memory Pools** tab in the Runtime Summary view displays detailed data and summary data about memory usage. The chart displays the available size of the Java heap (allocated memory) and the memory in use by your program over time. You can adjust the interval for data display (that is, updates to heap, non-heap, and garbage collection data), as well as control whether the chart displays heap or non-heap data. A table below the chart provides data for each memory pool.

Interpreting the Memory Pools Graph

The Memory Pools graph helps you to visualize how memory is being used in the heap. As you run your sessions, you will observe some common behaviors in the chart, including: a steady graph, increasing growth, growth and a drop, and spikes.

Note You may notice a growth in the heap when you start recording data with some JVMs. This is a result of engaging the JVM’s profiling interface. You may wish to adjust your heap size to accommodate this growth.

Steady graph

A steady graph indicates that some core set of objects are remaining in memory. You need to assess whether this is expected behavior or symptomatic of a problem. Refer to your program’s architecture for a list of persistent objects. A good understanding of

your long-lived objects helps you to determine if they are holding short-lived objects longer than necessary.

Increasing growth

Increasing growth in the Memory Pools chart is a clear indication of object allocations. While you are recording data, you expect objects to be allocated. Unless your hypothesis is that the objects will persist, you should expect these objects to be removed at the end of the session. A graph that does not drop to pre-recording levels, particularly after requesting one or more garbage collections, warrants further investigation. A large number of allocations may also prompt you to investigate the efficiency of the method; you may be allocating more objects than necessary.

Starting an application server can cause an initial growth in the heap. The memory used by the server is largely out of your control. It is worthwhile, however, to set the size of your original Java heap to be large enough to accommodate the server startup, rather than forcing the virtual machine to grow the size of the Java heap dynamically during the startup phase.

Growth and drop

Increasing growth followed by a large drop represents multiple allocations of short-lived objects over time that are cleared all at once. When the short-lived objects fill up the heap, the garbage collector frees memory.

Spikes

Steep spikes in the Memory Pools chart represent short-lived objects being allocated and garbage collected. Frequent garbage collections may cause your program to run more slowly. For more information, see “Tuning Garbage Collection and Finding Over-allocations” on page 117.

GC Data Tab

The **GC Data** tab displays garbage collection data in graphic and table formats. The Garbage Collections graph displays a color-coded bar for each type of collection and indicates the time that it occurred. The total number of GCs per interval is also indicated. A table in this view lists garbage collections by collector and indicates for each one the number, the length of time it took, as well as the cumulative number and time. The overhead for all GCs, expressed as a percentage of the total time the application ran, is indicated above the table.

Heap Data Tab

The **Heap Data** tab is similar to the Instances view (see "Exploring the Instances View" on page 138). It contains a horizontal bar chart and a table with information on the instances (allocated objects) in the Java heap. This includes the number of instances of each class in the heap at any given time and the memory used by those instances. The top row in the table contains totals for each of the columns. This table is not populated with heap values unless you have clicked the Refresh Runtime Data button.

Note The **Heap Data** tab is not available if you have connected to a JVM dynamically, using **Attach to Session > JMX** or `jpattach`.

Interpreting Heap Data

The horizontal bar chart illustrates the percentage of instances in each of the categories that are listed in the legend to the right of the chart. Hovering over a segment of the chart displays a popup showing the count or memory data for that category.

Each row in the table represents a class that has instances in the heap or has had all of its instances garbage collected. The Count columns tell you how many instances of a class currently exist in the heap, while the Memory columns show how much memory those instances consume. The Dead Count and Dead Memory columns show values for garbage collected instances that were allocated since recording started.

Filtering Heap Data

The Name Filter field allows you to reduce the set of displayed instances to only those that match the provided name. You can use the filter any time during a running session. When you filter the data, an indicator beside the filter displays the number of classes displayed out of the total number. For details about filter syntax, see "Syntax for Instance Filters" on page 142.

Note The memory value in the Totals line of the Runtime Summary table may not match the memory usage reported in the Instances view because of the differences in how the memory is tracked for each. For your investigation, you should focus on the values in the Instances view. For more information, see "Memory Troubleshooting Tips" on page 212.

4

Running a Memory Analysis

The most common reason to analyze memory usage is to find memory leaks. In Java, a memory leak refers to objects that loiter in the Java heap because they are not available for garbage collection. Another reason to analyze memory usage is to improve the efficiency with which the garbage collector runs.

This chapter describes some of the causes of loitering objects. It also describes how to run an analysis session, locate and investigate loitering objects, and presents some techniques for solving memory problems. The procedures are general, but the screenshots and examples are taken from the *LeakExample.class* program installed in the *<jprobe_home>\demos\memory\leakexample* directory.

This chapter contains the following sections:

Finding Memory Leaks 108
Tuning Garbage Collection and Finding Over-allocations 117

Finding Memory Leaks

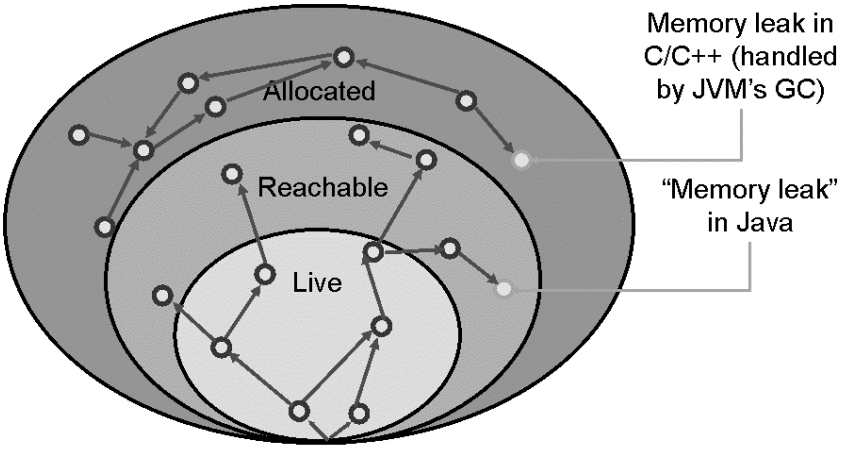
This section discusses memory leaks in terms of loitering objects and walks you through a basic Memory analysis, looking for loitering objects.

This section includes the following topics:

- "Understanding Loitering Objects" on page 108
- "Setting the Recording Level" on page 110
- "Performing a Basic Memory Analysis" on page 112

Understanding Loitering Objects

The key to fixing and avoiding loitering objects is good reference management. Loitering objects are objects in the heap that are reachable through references from the root set, but have no future value to the program. They are caused by a live object inadvertently or unnecessarily retaining a reference to the loiterer for some period of time. To allow the garbage collector to remove a loitering object—and possibly further instances that it references—you must remove all references to the loiterer.



Some objects are reachable from the *root set*, while others are not. A *root* is an initial entry point to the Java heap; the *root set* includes all entry points to the heap for your program. There are several ways that objects can be part of the root set, including but not limited to being one of the following:

- On the stack as a local variable
- Used for synchronization
- Referenced by JNI code
- A system class
- Referenced by a thread

The objects connected to the root set are either live or loitering—you need to determine if the objects have future value to your program. The unconnected objects no longer have any references and may be ready for garbage collection.

The loiterers that are easiest to detect are those that are allocated to the heap but not removed after the program is finished with them; these loiterers remain for the duration of the program's execution. More difficult to find are loiterers that are eventually freed or reused, but in the meantime consume memory that your program could be using for other tasks.

Four common causes of loitering objects are:

- Obsolete Collection References
- Obsolete Listeners
- Lingering Transitional References
- Stalled Stack References

Obsolete Collection References

An obsolete collection reference occurs when an object is added to a collection, but not removed when no longer required. When methods add objects without removing them, the collection can grow without bounds over the runtime of the program. You can avoid creating loitering objects by pairing add/remove calls.

Obsolete Listeners

Obsolete listeners are a type of obsolete collection reference. However, they warrant their own category because they are a common cause of loitering objects. Obsolete listeners also have a greater impact on the performance of your program because, as your listener collections grow, your program has to propagate events to more and more listener objects.

Lingering Transitional References

Long-lived objects often contain one or more transitional references, that is, references that will be reassigned to new objects when the state of the long-lived object changes. There are two situations to consider.

In the first case, a transitional reference holds onto its object (lingers) until the method that sets the reference is called again. For example, consider a print service that has a reference to the current print job. The job prints, but when it is complete, the reference is not freed, so the print job remains in memory until the next time you want to print something. You can avoid creating these kinds of loitering objects by setting the transitional reference to null when the object is no longer required.

In the second case, a singleton object changes state, but one of its transitional references is not updated. For example, consider an object that maintains information about files in a directory and has references to the largest, smallest, and most complex files. When you change directories, for some reason only the references to the largest and smallest files are updated—the reference to the most complex file still points to the file in the previous directory. This is really a bug, but one that is hard to track down. If you approach the problem as a loitering object, you may find it more quickly. Maintenance of singleton objects can be made easier by encapsulating state transitions in a single method, rather than scattering changes of state throughout your code.

Stalled Stack References

Long-running methods and stalled threads can pin objects in the heap until the method or thread returns. For example, consider the *StalledStack.class* application in the *demos* directory. The demo reads an XML file using DOM, parses out Java file names, and then performs a long-running (dummy) process on the parsed file names without freeing the memory taken up by the DOM tree. The memory will eventually be freed when the long-running process completes, but in the meantime, memory that could be used for the process is unavailable.

Setting the Recording Level

Before you start a Memory analysis, you need to decide what kind of data you want to see. The data recorded during the session provides you information about “Where did this object come from?”. Knowing where an instance came from can often help you track down a cause and resolution for a memory leak.

When dealing with a memory leak, you would be mostly interested in "Why is this object live?", information that can be determined by examining the referrers in the heap. For additional information about the instances (for example, when or where these instances were allocated), you must turn on the data recording. Enabling the recording for those instances is recommended when tuning the garbage collection and when analyzing over-allocation issues.

The *Set Recording Level* dialog box presents the following options.

Select this option	In this situation
Data Recording Off	You are at the initial stage of your investigation, when you do not know what is leaking or why. This is often enough to at least start your investigation. **Note** Recording should also be Off while your application and/or application server is starting, unless you need to record the startup behavior.
Record Allocations For All Instances	You want to record only the allocation times of instances, which are displayed in the *Instance Detail* view. The *Call Traces* and *Merged Allocation Points* views are not available in this case, as no trace data is collected. **Note** If you have a very large program, if you are using an application server, or if you know what class of objects you want to track, consider using *Record Data as Specified in the Table Below*, because *Record Allocations For All Instances* may have a severe performance and memory impact.
Record Allocations and Stack Traces For All Instances	You want to record allocation times, as well as stack traces. You can use the *Call Traces* and *Merged Allocation Points* views to analyze "where instances came from". **Note** If you have a very large program, if you are using an application server, or if you know what class of objects you want to track, consider using *Record Data as Specified in the Table Below*, because *Record Allocations and Stack Traces For All Instances* may have a severe performance and memory impact.

Select this option	In this situation
Record Allocations, Stack Traces, and Garbage Data For All Instances	You want to find what methods are allocating short-lived objects in your program, or where these objects were allocated (that is, you want to use the *Call Traces* or *Merged Allocation Points* views while investigating by *Dead Count* or *Dead Memory*). **Note** If you have a very large program, if you are using an application server, or if you know what class of objects you want to track, consider using *Record Data as Specified in the Table Below*, because *Record Allocations, Stack Traces, and Garbage Data For All Instances* may have a severe performance and memory impact.
Record Data as Specified in the Table Below	You know the class or classes of objects that you need to track. This option can be particularly useful with large programs where collecting detailed trace or garbage data for all instances is slow and uses a lot of memory. **Note** If you are trying to use one of the other recording options (other than *Data Recording Off*) and your program is very slow or you receive errors from JProbe about being out of memory, you may need to use this option.

Note Unless the Data Recording is off, a new recording session always begins after you click **OK**, even if you did not change the level.

Performing a Basic Memory Analysis

Before you can run a Memory analysis, you need a settings file and a script file that starts your application. For a description of how to produce these files, see Configuring JProbe to Analyze Your Application.

The next steps in a JProbe Memory analysis are to start your application and then connect to it from the JProbe Console. You can then analyze the data that JProbe collects and displays in the Runtime Summary view.

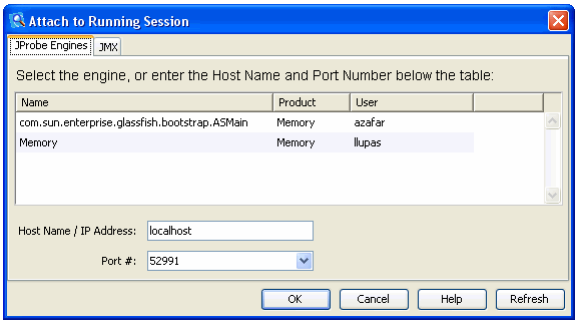
To run a JProbe Memory analysis:

1 From the command line, run the batch file that was created when you created your settings file.

Note This procedure offers only an example of how you can run a Memory analysis using JProbe. The actual steps may vary depending on your specific application.

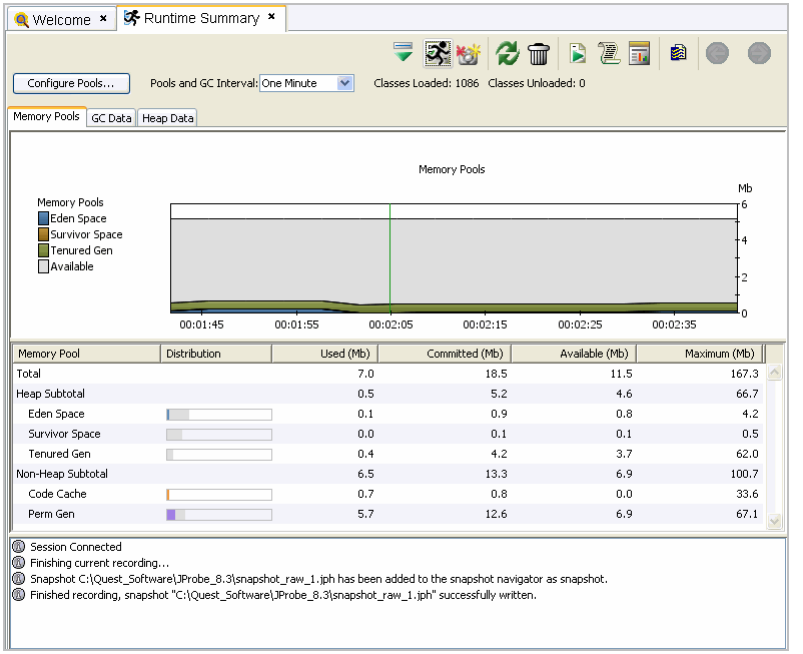
2 In JProbe, click **Attach to Session** on the toolbar.

The **Attach to Running Session** dialog box appears.



3 Select the appropriate engine from the list and click **OK** to connect JProbe to your application.

The *Runtime Summary* view appears. For a description of this view, see "Exploring the Memory Runtime Summary View" on page 132.

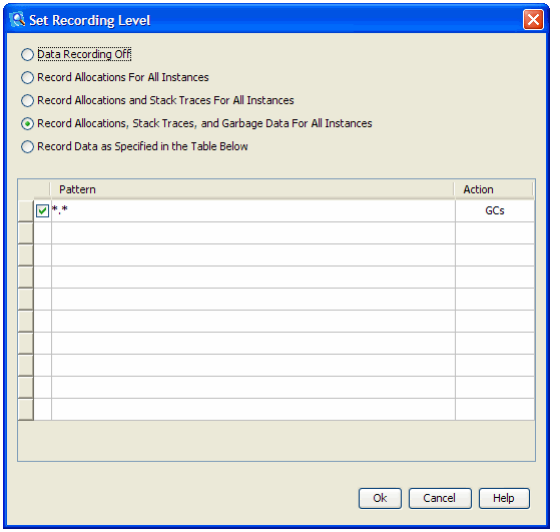


4 Before proceeding with the Memory analysis, you can choose at this point to explore the data and ask a few questions, such as “is my application using about the right amount of memory?”. If the application uses too much memory, you could then identify what objects are live (using the **Heap Data** tab on the *Runtime Summary* view), then decide whether you want to proceed by taking a snapshot (to answer the question “why is this live?” and include a heap dump) or by beginning recording (to answer the question “where do these come from?”). This helps you decide whether you need to use some filters for your recorded data.

5 In the *Runtime Summary* view, click the **Data Recording** icon .

The *Set Recording Level* dialog box appears.

Note The *Set Recording Level* dialog box is always initialized to the value you have last set. To see the settings in effect in the JProbe Analysis Engine, check the Session History (for details, see “Viewing the Session History” on page 445).

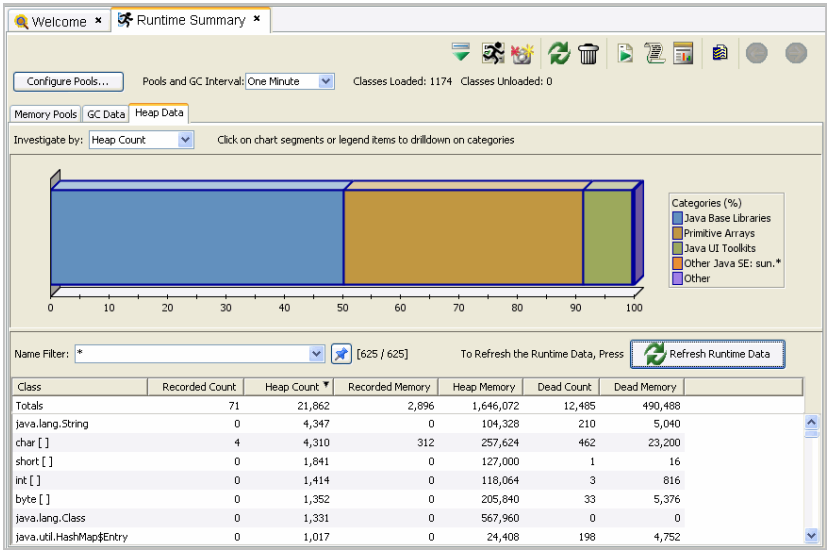


6 If you want to change the Recording Level, select one of the available options (for details, see “Setting the Recording Level” on page 110) and click **OK**.

If you were collecting data from startup, this data is written to a snapshot. If you are happy with the current recording level and want to exit without writing a snapshot, click **Cancel**.

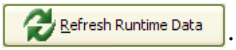
7 To take a snapshot during the session, click **Take a Snapshot** .

8 On the *Runtime Summary* view, click the **Heap Data** tab.


Welcome
Runtime Summary
Configure Pools...
Pools and GC Interval: One Minute
Classes Loaded: 1174 Classes Unloaded: 0
Memory Pools
GC Data
Heap Data
Investigate by: Heap Count
Click on chart segments or legend items to drilldown on categories
Categories (%)
Java Base Libraries
Primitive Arrays
Java UI Toolkits
Other Java SE: sun.*
Other
0 10 20 30 40 50 60 70 80 90 100
Name Filter: *
[625 / 625]
To Refresh the Runtime Data, Press
Refresh Runtime Data
Class | Recorded Count | Heap Count | Recorded Memory | Heap Memory | Dead Count | Dead Memory
Totals 71 21,862 2,896 1,646,072 12,485 490,488
java.lang.String 0 4,347 0 104,328 210 5,040
char [ ] 4 4,310 312 257,624 462 23,200
short [ ] 0 1,841 0 127,000 1 16
int [ ] 0 1,414 0 118,064 3 816
byte [ ] 0 1,352 0 205,840 33 5,376
java.lang.Class 0 1,331 0 567,960 0 0
java.util.HashMap$Entry 0 1,017 0 24,408 198 4,752

This view is similar to the *Instances* view (see “Exploring the Instances View” on page 138), except for the **Refresh Runtime Data** button and the information below the table.

9 To display the data that has been collected, click **Refresh Runtime Data**



.

10 Click **Data Recording**  , select the *Data Recording Off* check box, and click **OK**.

JProbe turns the data recording off and takes a snapshot.

11 To end the analysis, click **Detach From Session**  on the JProbe Console toolbar.

The *Instances* view opens, displaying the data in the snapshot taken before detaching from the running session. You can now examine the data in various ways to identify loitering objects. For a detailed description of the *Instances* view, see “Exploring the Instances View” on page 138.

Tuning Garbage Collection and Finding Over-allocations

The discussion in "Understanding Loitering Objects" on page 108 explains how your program's runtime performance can be adversely affected by the garbage collector. In addition to eliminating loitering objects with Memory and resolving performance bottlenecks in your code with Performance, you can assess the behavior of the garbage collector with your program.

Tuning garbage collections is dependent on the JVM that you are using to run your program. Although garbage collectors all perform the same basic function, the way they operate varies widely among JVMs and even within a JVM. JProbe can help with some general analysis, and some tasks that are common, but for complete tuning you will also need to consult your JVM vendor's documentation.

A closely related issue is excessive object allocation. JVMs have many optimizations for allocating small, inexpensive objects, but objects that are expensive to allocate, even if they do not cause garbage collection problems, can hurt performance.

Understanding Garbage Collection Issues

Garbage collection will always have some impact on your program, and the type of garbage collector affects the impact. Some garbage collectors can cause long pauses that are noticeable (and annoying) in programs with graphical user interfaces, but acceptable in server applications. Others minimize the pause time, but this can lower the throughput of server applications.

Modern JVMs use "ergonomics" to try to adapt garbage collector behavior automatically to work best with your program. They may allow you to specify a goal (such as low pause times or high throughput) and will pick an appropriate garbage collection strategy. Often this automatic behavior is good enough, but you can use the Runtime Summary view to help you decide if you need to do further tuning. (See "Monitoring GC Overhead with the Runtime Summary View" on page 118 for details.).

One key to achieving good GC behavior that is common to all JVMs is correct sizing of the Java heap. Many collectors are "generational" garbage collectors, because they divide the heap into pools for newly allocated objects and older objects that use different garbage collection strategies. Getting the relative sizes of the generations right is also important for good overall garbage collector performance.

Monitoring GC Overhead with the Runtime Summary View

Garbage collection overhead is defined as the percentage of time the garbage collector is running compared to the overall elapsed time of your program. A low number means garbage collection is having low impact on your program. A high number can indicate a problem.

When you are attached to a running session, JProbe shows the Garbage Collection Overhead below the Garbage Collections chart in the **GC Data** tab of the *Runtime Summary* view.

The table in the **GC Data** tab also shows a breakdown by garbage collector. The names of the garbage collectors are provided by the JVM; consult your vendor documentation for details. JProbe shows each garbage collector, the time it has used, and the number of times it has been invoked.

Collecting Data for GC Tuning with the Runtime Summary View

If you need to tune garbage collection, you will need to consult documentation from your JVM vendor about how to do this. This documentation will usually suggest using the `-verbose:gc` option to collect data for the tuning, but this output can be very difficult to understand. The Runtime Summary view shows similar data in a format that can help you understand the current garbage collection behavior and heap sizing to help you tune.

Start your application with JProbe and let it get to its normal operating state. During startup the program behavior is different as many long-lived objects are allocated for the first time, and the garbage collector is adjusting to the program. After your application is running normally, the Runtime Summary view shows the typical operating behavior.

The Runtime Summary view's **Memory Pools** tab displays heap sizes and garbage collection behavior. The different pools are named by the JVM, and your JVM vendor documentation may explain what they represent. Usually, they reflect the different generations of a generational garbage collection scheme.

The gauges in the lower section of the **Memory Pool** tab (in the *Distribution* column) show the current state of each pool. The full length of the gauge represents the maximum size of the pool, which may have been set directly or indirectly by command-line options to the JVM (for example, `-Xmx512m`). The colored segment of the gauge represents an estimate of the amount of memory currently in use in the pool, called Used. The gray segment of the gauge represents memory that the JVM has committed from the operating system, but is not yet occupied. JProbe calls this "Available". The actual values for the Used, Committed, Available, and Maximum memory are displayed in the columns to the right of the gauges.

The chart in upper section of the **Memory Pools** tab shows a history of the Used portion of each of the pools, along with the combined Available memory from all pools.

Many of the issues your vendor tuning documentation discusses are often seen clearly in this view. For example, if your young generation is too small, or objects are being moved to the old generation too soon, this will show in the view. The young generation will fill often, and the objects will be tenured to the old generation so the gauges for the pool or pools in the young generation will fill and empty regularly. Since short-lived objects will have been moved to the old generation too soon, you will see it regularly growing with the new objects, then shrinking as it garbage collects them.

Over-allocation of objects may also be visible as spikes of Used memory in one or more of the pools in the Memory Pools chart.

Reducing Over-allocation of Objects

Over-allocation of objects can have a performance impact either by affecting GC overhead or if the objects are very expensive to create (for example, database connection objects). If you suspect over-allocation may be a problem, JProbe can help find the source code that is causing the allocations, so that you can correct it.

It is important to note that over-allocation is not always a problem. JVMs are tuned to optimize allocation, particularly of short-lived objects. Some techniques used to reduce allocation, such as object caching, can actually have a negative effect on garbage collection, so it is important to consult vendor documentation and to measure to see if your changes do improve performance.

As mentioned above, over-allocation may be visible in the Runtime Summary as frequent steep spikes in the Memory Pools chart.

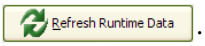
To determine the source of the allocations:

1. In the *Runtime Summary* view, click **Data Recording**  .

   The *Set Recording Level* dialog box appears.

2. Select *Record Allocations, Stack Traces, and Garbage Data For All Instances* and click **OK**.

3. Exercise a use case in your application to generate activity in the JVM heap.

4. Click the **Heap Data** tab of the *Runtime Summary* view, and click **Refresh Runtime Data** 

.

   It may take several seconds for data to be collected from the JVM.

5 When the table has been populated with data, select *Dead Count* or *Dead Memory* from the *Investigate by* list.

The bar at the top of the view shows a breakdown of the objects that have been allocated and garbage collected during data recording. You can drill down by clicking on segments of the bar, or you can sort the table by Dead Count and look for classes your code allocates. You can also filter the table to help you narrow the search. (For details, see "Filtering Data in the Instances View" on page 141.)

Often identifying a class or classes that are being allocated more than you expect is enough for you to know where to look in your program to change this behavior. If not, you can use JProbe to help you find the problem code.

6 To get a snapshot of the data that JProbe has recorded, click **Take a Snapshot** 

in the *Runtime Summary* view or click **Data Recording**  and select *Data Recording Off*.

A new memory snapshot is added to the Snapshots Navigator.

You can open this snapshot in the *Instances* view by right-clicking the snapshot and clicking **Open Snapshot**. Select *Dead Count* or *Dead Memory* from the *Investigate by* list, then and select from the table the class you found that is being over-allocated.

Examining Garbage Collection Data

Two views are especially useful for finding a problem: the Call Traces view and the Merged Allocation Points view. Both views show the call traces for garbage collected objects, but from different perspectives.

The Call Traces view shows the data in a traditional Call Graph or Call Tree, much like the Call Graph in JProbe Performance, except with object allocation data instead of times. You may find this view more helpful in understanding the overall allocation behavior.

The Merged Allocations Point view shows the data starting from the allocating method. This view shows the merged allocation traces for a set of objects, which may be more helpful in quickly identifying the cause of a problem.

Using the Merged Allocation Points View

The Merged Allocation Points view works almost the same way with garbage collection traces as it does with other recorded data (see "Exploring the Merged Allocation Points

View" on page 181), except that all the objects have been garbage collected, so you cannot go to the Instance Detail view.

Using the Call Traces View

The Call Traces view works almost the same way with garbage collection traces as it does with other recorded data (see "Exploring the Call Traces View" on page 174), except that all the objects have been garbage collected, so you cannot go to the Instance Detail view.

5

Learning More About Memory Options

This chapter presents how to fine-tune the Memory analysis tool and how to analyze a heap dump.

This chapter contains the following sections:

Analysis Options for Memory....124
Importing Heap Dumps....127

Analysis Options for Memory

You can configure the following data collection settings for your Memory analysis:

- Filters for Memory
- Triggers for Memory
- Initial Recording for Memory

By default, JProbe collects allocation data only. You can collect stack traces and data on garbage collections by setting filters.

Filters for Memory

You can select *Data Collection Filters* for your Memory analysis in the **Initial Recording** tab of the configuration wizard.

Unlike the other analysis tools, you should avoid excluding code from a JProbe Memory analysis during early stages of your analysis. If you do exclude code, you may not see object allocations that are important to the analysis.

Note In the JProbe Console, you can select filters via the Set Recording Level dialog. The filters table in the dialog contains a default filter to track all allocations made by your application. Running a session with this default filter helps you to identify problem classes in your application.

If you know which classes are causing your problems, you can set up JProbe to collect more data on them. To collect stack traces as well as allocation data, define filters for the problem classes and set the action to *Include*.

Caution Collecting detailed information on stack traces and garbage collections requires a lot of resources. Set these filters on problem classes only, not your entire application.

Filters are defined in the same way for all the analysis tools, however the filter actions differ among the analysis tools. This section summarizes the filter actions for a Memory

analysis. To learn how to create filters, see Refining Data Collection Filters. Filters for a Memory analysis can be set on a package or on a class.

Filter Action	Description
GCs	JProbe collects data about object allocations, stack traces, and garbage collected instances for the specified package or class. In the JProbe Console, data collected with this action is displayed in the Merged Allocations Points view and the Call Trace view when investigating by Dead Count/Recorded Memory.
Traces	JProbe collects data about object allocations and stack traces for the specified package or class. In the JProbe Console, data collected with this action is displayed in the Merged Allocations Points view and the Call Trace view when investigating by Recorded Count/Memory.
Allocations	*Default*. JProbe collects the following data about object allocations for the specified package or class: • Allocation time for each instance • Whether an instance was garbage collected • Recording boundary for instances that remain live when a snapshot is taken
Exclude	JProbe does *not* collect any data for the specified package or class.

Triggers for Memory

Triggers provide a way to tie JProbe actions to your program's execution. You can tell JProbe to perform an action when your program enters or exits a method of your choice. With advanced trigger controls, you can also specify whether to start executing the trigger actions at method entry or exit. Triggers are useful for automated sessions.

Triggers are defined in the same way for all the analysis tools, however the trigger actions are somewhat different. This section describes the trigger actions for a Memory analysis. To learn how to create triggers, see “Setting Triggers” on page 425.

Trigger Action	Description
Snapshot	Takes a snapshot. The snapshot file name takes the form *snapshot_<n>_raw.jph*.
Recording	Starts recording data. In the Recording Parameter section, you can select the level of data you want to record: • Data Recording Off • Record Allocations For All Instances • Record Allocations and Stack Traces For All Instances • Record Allocations, Stack Traces, and Garbage Data For All Instances • Record Data as Specified in the (filters) Table Below **Note** When the trigger action is set to *Recording*, the selected recording parameter is also displayed in the triggers table.
Request GC	Requests a garbage collection.

Initial Recording for Memory

You can select initial recording options for your Memory analysis in the **Initial Recording** tab. The following *Initial Recording at JVM Start* options are available for a Memory analysis:

- For Java Applications:
  - Data Recording Off (default selection)
  - Record Allocations For All Instances
  - Record Allocations and Stack Traces For All Instances
  - Record Allocations, Stack Traces, and Garbage Data For All Instances
  - Record Data as Specified in the Table Below
- For all supported Application Servers:
  - Data Recording Off (default selection)
  - Record Data as Specified in the Table Below

Importing Heap Dumps

JProbe is a powerful tool for analyzing heap dumps. You can import a heap dump and save it as a memory snapshot. However, imported heap dumps lack some of the data that is available in a JProbe snapshot, such as traces and garbage collection data. Some imported dumps do not contain field names. Also, PHD and text dumps do not contain any information on roots, so it is sometimes difficult to tell if an instance is actually pinned.

- "Importing an IBM JVM Heap Dump" on page 127
- "Importing an HPROF Heap Dump" on page 129

Importing an IBM JVM Heap Dump

If you experience an out-of-memory exception, or you want to take a snapshot of the IBM JVM outside of JProbe, you can generate a heap dump. You can then import it into JProbe and view it as a snapshot. JProbe supports two different formats for importing heap dumps: IBM Text dump (.txt) or IBM Portable Heap Dump (.phd). For more information, see How to generate a heap dump from an IBM JVM.

The heap dump information that is saved to the snapshot contains classes and instances with their associated references and referrers. Some objects in the heap dump have a path to a root and others have zero referrers but are not declared as roots. Because neither text nor PHD format provides root information, it is sometimes difficult to tell whether an instance is actually pinned or if it is ready to be garbage collected. Therefore, to provide a complete list of all objects in the heap dump, objects that have paths to a root and non-rooted objects are all included in the snapshot.

To import an IBM dump:

1. Select **File > Import**.
2. Select one of the following options:
   - Import IBM Text File
   - Import IBM PHD File
3. From the **Select IBM Text/PHD File** dialog box, browse to the file that you want to import.
4. Click **Open**.

   The dump is converted to a Memory snapshot.

5 Select and view the Memory snapshot as usual.

How to generate a heap dump from an IBM JVM

Some IBM JVMs automatically generate a heap dump when an out-of-memory exception is thrown. You can also generate a heap dump in situations where you are using an IBM JVM and are unable to take a snapshot. You need to set the environment variables to allow heap dumps on request and to specify the format of the heap dump (with the exception of the out-of-memory situation described below).

You can generate a heap dump by sending a signal to the process (**CTRL+** on UNIX and **CTRL+Break** on Windows). IBM_HEAP_DUMP and IBM_HEAPDUMP must be set before taking a heap dump.

The following environment variables allow the JVM to generate a list of all the objects in the Java heap.

IBM Dump Task	Action

Add Java environment variables	On your system, set the environment variables:
	`IBM_HEAP_DUMP=true`
	`IBM_HEAPDUMP=true`
	`IBM_HEAPDUMP_OUTOFMEMORY=true`
	`IBM_JAVACORE_OUTOFMEMORY=true`
	`IBM_JAVA_HEAPDUMP_TEXT=true`*

*If you set the environment variable `IBM_JAVA_HEAPDUMP_TEXT=true,` the JVM creates a text heap dump. If you set this environment variable to `false,` the JVM creates a PHD heap dump. The default setting is `false`. Text dumps are usually much larger than PHD dumps. JProbe can import either format, but with some versions of the JVM, PHD dumps are missing information about array types.

Optionally, you can set the `IBM_HEAPDUMPDIR=<directory>` environment variable. This environment variable controls the location where the heap dump file is saved.

Starting with SDK 1.4.2, some variables automatically set by the JVM (`IBM_HEAPDUMP_OUTOFMEMORY` and `IBM_JAVACORE_OUTOFMEMORY)` can result in several heap dump files (heapdump##.txt) and Javacore files (javacore##.txt) being generated when the Java heap becomes exhausted. This can be disabled by setting the following environment variables:

```
IBM_HEAPDUMP_OUTOFMEMORY=false
IBM_JAVACORE_OUTOFMEMORY=false
```

Importing an HPROF Heap Dump

Most JDKs include the HPROF utility, a simple command-line tool for profiling your application's heap or CPU usage. JProbe supports importing a HPROF heap dump in binary and text formats. This may be useful if you use the `jmap` or `jconsole` tools in the JDK to produce heap dumps, or if you use the HPROF utility for other purposes in your development environment and would like to be able to view the heap dump information that you collected in a graphical interface. The heap dump is converted to a snapshot that you can analyze using the JProbe Memory views.

To import a binary HPROF dump:

1. Select **File** > **Import**.
2. Select one of the following options:
   - Binary HPROF File
   - Text HPROF File
3. From the **Select Binary/Text HPROF File** dialog box, browse to the file that you want to import.
4. Click **Open**.

   The dump is converted to a Memory snapshot.
5. Select and view the Memory snapshot as usual.

How to generate a heap dump from a Sun JVM

Some Sun JVMs (Sun Java 6 JVMs and the latest releases of 1.4.2 and Java 5 JVMs) can be configured to generate a HPROF format heap dump when an out-of-memory exception is thrown. To generate a dump automatically, add the `-XX:+HeapDumpOnOutOfMemoryError` option to the `java` command line when you launch your program. If your program encounters an `OutOfMemoryError`, the JVM attempts to write an HPROF format heap dump to a file named *java_pidNNNN.hprof* in the current directory, where NNNN is the ID of the `java` process. You can import this file into JProbe.

You can also use the `jmap` tool with the `-dump:live,format=b` option to produce a dump. See the `jmap` documentation for details.

6

Exploring Memory Analysis Views

This chapter provides detailed information about the Memory views used to present memory data.

This chapter contains the following sections:

Exploring the Memory Runtime Summary View 132
Exploring the Instances View 138
Exploring the Dominating Instances View 148
Exploring the Pinned Instances View 155
Exploring the Allocations View 160
Exploring the Allocations Detail View 163
Exploring the Instance Detail View 166
Exploring the Call Traces View 174
Exploring the Merged Allocation Points View 181
Exploring the Heap Graph View 185
Exploring the Leak Doctor View 190
Exploring the Memory Source View 196
Exploring the Memory Difference View 197
Setting Preferences for Memory Views 202

Exploring the Memory Runtime Summary View

A Memory analysis helps you to identify references that are holding on to loitering objects and to identify the reason for this. As you run a Memory session, the Runtime Summary view displays instance counts and changes in counts in the Java heap. After you capture the data in a memory snapshot, you can use the heap investigation tools, starting with the Instances view, to understand the references among classes in your program.

This section contains the following information:

- Understanding the Memory Runtime Summary View
- Interacting with the Memory Runtime Summary View

Understanding the Memory Runtime Summary View

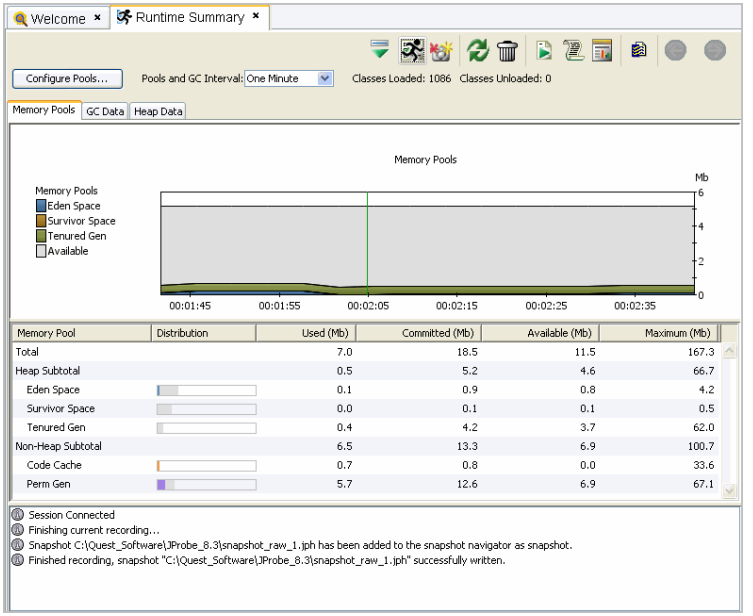
The Memory Runtime Summary view is both a view of the Java heap and a control center for running your analysis session. It helps you to identify loitering objects, excessive garbage collections, and over-allocation of objects, and also acts as a progress bar. A green vertical line on the Runtime Summary chart indicates that a snapshot was taken, either due to a change in recording level, or because a snapshot was requested.

After you have attached to a running session of your application, the Runtime Summary view appears. It has three tabs:

- The Memory Pools Tab displays memory usage in graphic and table formats.
- The GC Data Tab displays garbage collection data in graphic and table formats.
- The Heap Data Tab displays information about the instances of classes in graphic and table format.

Toolbar buttons provide access to the snapshot and garbage monitoring controls.

The system messages area (bottom pane) displays any errors and warnings that occur (including those that have occured before the this Console connected to the JProbe Analysis Engine).



Memory Pools Tab

The Memory Pools chart shows the history of the used and available values of the pools. The Available part is the sum of the displayed pools' Available memory.

While you are running an analysis session, the summary graph displays the size of the heap memory pools by default. Heap pools contain things normally found in the Java heap, like Java objects. Non-heap pools normally contain things the JVM uses internally, such as the compiled representation of method code or class information. To change from viewing heap to viewing non-heap data, or vice versa, use the **Configure Pool** button.

You can choose a different time interval for the summary graph from the *Pools and GC Interval* list. The options are One Minute, Five Minutes, Fifteen Minutes, Thirty Minutes, and One Hour.

The data that is displayed in the Memory Pools graph varies, depending on the JVM and the options that you are using. For JVMs that use generational garbage collectors, heap pools typically represent the generations. For example, in Sun JVMs, with some options (as shown above), Tenured Gen is the old generation, while Eden Space and Survivor Space together make up the new generation.

The memory usage of the pool is an estimation provided by the JVM. It consists of three values, which appear in tooltips when you hover over a gauge:

Used	The amount of memory currently being used.
Committed	The amount of memory that the OS has guaranteed to be available to the JVM. This will always be greater than or equal to Used.
Max	The maximum amount of memory that pool will use. Not all pools have a maximum value. Maximum value is often affected or directly set by JVM options like `-Xmx`. A maximum value will always be greater than or equal to the committed value.

JProbe defines Available Memory as the difference between committed and used.

GC Data Tab

The **GC Data** tab contains a graph and a table. The graph displays each type of garbage collection at the time it occurred. The Count ticks at the right of the graph indicates how many collections were done of each type over an interval.

The Garbage Collection Overhead data below the graph indicates the percentage of time the JVM was exclusively performing garbage collection compared to the elapsed time the program has been running. For example, a program that has been running for 10 seconds and has spent 0.5 seconds doing garbage collection has a GC overhead of 5%. The lower the percentage, the lower the overhead. If the percentage is high, you may need to tune your JVM's GC options or reduce object cycling.

The table provides a breakdown by garbage collector that shows the number of times each collector has been invoked and the time (in milliseconds) that it used, as well as the cumulative counts and times.

Heap Data Tab

The **Heap Data** tab is similar to the Instances view (see "Exploring the Instances View" on page 138). The main differences are the toolbar options and the **Refresh Runtime Data** button. The Instances view also allows more in-depth analysis. When you switch to the **Heap Data** tab during a session, be sure to click this button to ensure that you are seeing the latest data.

Interacting with the Memory Runtime Summary View

The following table summarizes the tasks that you can perform.

Task	Action	Response
Change the data recording level or add a filter.	Click **Set Recording Level**  to open the *Set Recording Level* dialog box, then make the desired changes.	JProbe collects the selected type of data or applies the filter specified in the table.
Take a snapshot	Click **Take Snapshot**  .	A snapshot of heap is taken, along with any data recorded. It appears in the Snapshots Navigator.
Display runtime data in the **Heap Data** tab	Click **Refresh Runtime Data**.	The table refreshes and displays the most recent runtime data.
Request a heap garbage collection	Click **Request Garbage Collection**  .	A garbage collection operation is requested of the JVM. Results can be observed in the **GC Data** tab.
View the execution log from the Runtime Summary view	Click **Runtime Execution Log**  .	The **Execution Log** dialog box opens and you can review the information by navigating through the available tabs.

Task	Action	Response
View the history of events since the Console connected to the session.	Click **View Session History**  .	The **Session History** dialog box opens. In the left pane, you can parse the list and review the events that happened since the Console last connected to the session. In the right pane you can view additional details for select events.
Create a report for this view	Click **Generate Report**  to open the *Create Runtime Summary Report* dialog box. Select a location for saving the report and click **Generate File**.	The report is generated in the specified location. For more information, see "Runtime Summary Report" on page 204.
Create a support bundle	Click **Create Support Bundle**  to automatically collect basic support information into a ZIP file, for submission to Quest Support.	The **Support Bundle Created** dialog box appears, indicating the location where the ZIP file was saved. You can click **Copy Bundle Name** to copy to the clipboard the name of the bundle that was just created.
Display non-heap data	Click the **Configure Pools** button to open the *Select Memory Pools* dialog box, then select the **Non-Heap** option. **Note** This action may not be available for all JVMs.	The graphs refresh to display the non-heap data. Information on other data structures that are occupying memory (such as the permanent generation), may be included if the JVM supports it.

Task	Action	Response
Change the data interval for the memory pools and garbage collections.	Select the appropriate option from the *Pools and GC Interval* list.	The Memory Pools and Garbage Collection graphs change to the selected interval.
Save the chart as a PNG or JPEG file	Right-click in the chart and select **Save Chart As**, then select the graphic file type (**PNG** or **JPEG**) in which you want to save the chart. In the *Export Image* dialog box, specify the export parameters and a location to save the file.	The file is saved to the location you indicate.
Export the chart to CSV	Right-click in the chart and select **Export to CSV**. In the *Export to CSV* dialog box, specify the export parameters and a location to save the file.	The file is saved to the location you indicate.
Show/hide table columns	Right-click and select **Show/ Hide Columns**. In the *Show/ Hide Columns* dialog box, select the columns to show and click **OK**.	The selected columns are displayed. The rest are hidden.
Resize column widths	To set the column width, drag the edge of the column heading left or right. To see all data in a column, right-click in the column and select **Size <column name> to Fit**. To fit all columns in the viewing area, right-click and select **Size All Columns to Fit**.	The columns are resized.
Rearrange columns	Drag a column by its heading and drop it in a new location.	The column appears in its new location.

Task	Action	Response
Sort the table by a metric	Click the column heading for the metric. To reverse the sort order, click the heading again.	The table is sorted.
Export the table to CSV	Right-click in the table and select **Export to CSV**. Type a file name and click **Save**.	The file contains the table data separated by a comma. It may also contain headers. For more information, see "Changing How JProbe Exports to a CSV File" on page 455.

Exploring the Instances View

The Instances view is the starting point for investigating a heap. It displays data for the entire Java heap and contains data on the instances of classes (objects) allocated during the execution of your program. You can apply filters to see a subset of the data.

This section contains the following information:

- "Opening the Instances View" on page 138
- "Understanding the Instances View" on page 139
- "Filtering Data in the Instances View" on page 141
- "Interacting with the Instances View" on page 144

Opening the Instances View

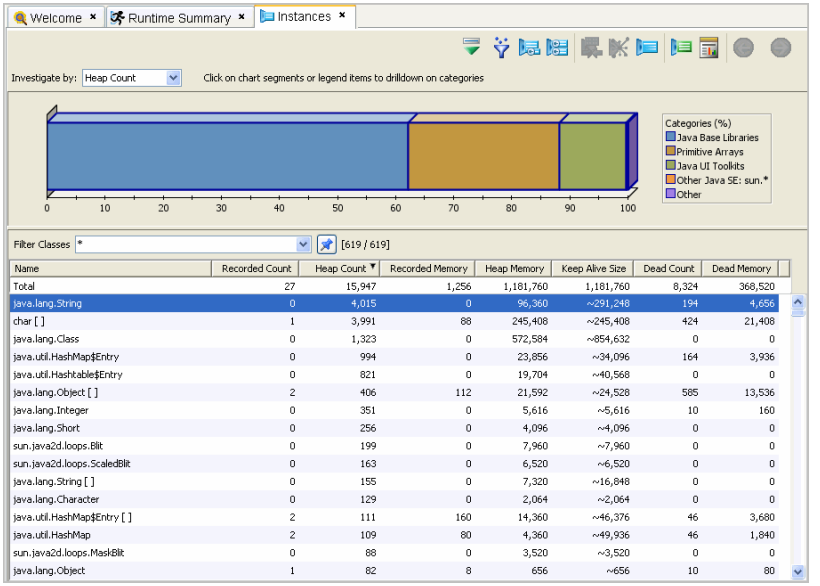
The snapshots that you took during a session are listed in the Snapshots Navigator.

To open the Instances view:

1. In the Snapshots Navigator, select the snapshot that you want to view.
2. Double-click the snapshot.

Understanding the Instances View

The Instances view consists of a horizontal bar chart and a table.



Name	Recorded Count	Heap Count	Recorded Memory	Heap Memory	Keep Alive Size	Dead Count	Dead Memory
Total	27	15,947	1,256	1,181,760	1,181,760	8,324	368,520
java.lang.String	0	4,015	0	96,360	~291,248	194	4,656
char [ ]	1	3,991	88	245,408	~245,408	424	21,408
java.lang.Class	0	1,323	0	572,584	~854,632	0	0
java.util.HashMap$Entry	0	994	0	23,856	~34,096	164	3,936
java.util.Hashtable$Entry	0	821	0	19,704	~40,568	0	0
java.lang.Object [ ]	2	406	112	21,592	~24,528	585	13,536
java.lang.Integer	0	351	0	5,616	~5,616	10	160
java.lang.Short	0	256	0	4,096	~4,096	0	0
sun.java2d.loops.Blit	0	199	0	7,960	~7,960	0	0
sun.java2d.loops.ScaledBlit	0	163	0	6,520	~6,520	0	0
java.lang.String [ ]	0	155	0	7,320	~16,848	0	0
java.lang.Character	0	129	0	2,064	~2,064	0	0
java.util.HashMap$Entry [ ]	2	111	160	14,360	~46,376	46	3,680
java.util.HashMap	2	109	80	4,360	~49,936	46	1,840
sun.java2d.loops.MaskBlit	0	88	0	3,520	~3,520	0	0
java.lang.Object	1	82	8	656	~656	10	80

The *Investigate By* list above the chart allows you to select which metric should be used to determine the top five categories. The following options are available:

- Recorded Count
- Recorded Memory
- Heap Count
- Heap Memory
- Dead Count
- Dead Memory

It is important to keep in mind that the option you select from this list affects not only the *Instances* view, but also the availability of subsequent views. If you cannot access a view or you do not see the data you expect to see in a subsequent view, it may be due to the *Investigate by* option that you select in the *Instances* view.

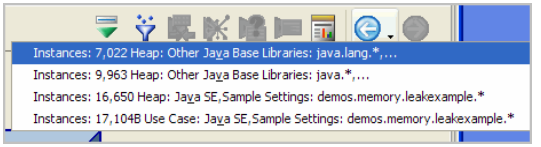
The following table lists some examples of how to use the *Investigate By* options.

If you want to:	Select this *Investigate By* option
See recorded data in subsequent views	*Recorded Count* or *Recorded Memory*
See heap data in subsequent views	*Heap Count* or *Heap Memory*
See garbage data in subsequent views	*Dead Count* or *Dead Memory*
Investigate traces for dead objects	*Dead Count* or *Dead Memory* and go to the *Call Traces* or *Merged Allocation Points* view
See instances created during your session in the Instance Detail view	*Recorded Count* or *Recorded Memory* and go to the *Instance Detail* view
Consider all instances in the heap, instead of just those watched while recording	*Heap Count* or *Heap Memory*

Some views only display certain types of data. For example, the Merged Allocation Points and Call Traces views display only recorded data or garbage collection data. You will not be able to access these views when Investigate By is set to Heap Count or Heap Memory, or when the chosen class does not have any recorded or garbage instances.

The bar chart in the Instances view illustrates the percentage of instances in each of the categories that are listed in the legend to the left of the chart. Hovering over a segment of the chart displays a popup showing the number of instances in or memory used by that category.

You can drill down in a category by clicking either a section in the chart or the corresponding category in the legend. As you drill down to more detailed information, the chart and table refresh to show only the instances in that category. You can click the back  and forward  toolbar buttons to move backward and forward through operations you have already performed, or you can click the down arrow beside either button and go directly to any of the other operations that you have performed.


Instances: 7,022 Heap: Other Java Base Libraries: java.lang.*,...
Instances: 9,963 Heap: Other Java Base Libraries: java.*,...
Instances: 16,650 Heap: Java SE,Sample Settings: demos.memory.leakexample.*
Instances: 17,104B Use Case: Java SE,Sample Settings: demos.memory.leakexample.*

You can sort the table by any of the columns. The text field above the table allows you to filter the classes that are listed in the table. See "Filtering Data in the Instances View" on page 141 for details.

Filtering Data in the Instances View

When you adjust filters in the Instances view, you restrict the number of classes that can be displayed. There are two ways to filter the data in the Instances view:

- You can use the Filter Classes text box above the table to filter by class, which narrows what is displayed in the view, but does not affect the underlying data set.
- You can also use the Set Filters dialog to set one or more filters by class. Using this method narrows or expands the underlying data set.

To filter the Instances view by class:

1. Type a pattern or comma-separated list of patterns in the Filter Classes text box.
2. Press **Enter** to apply the filter.

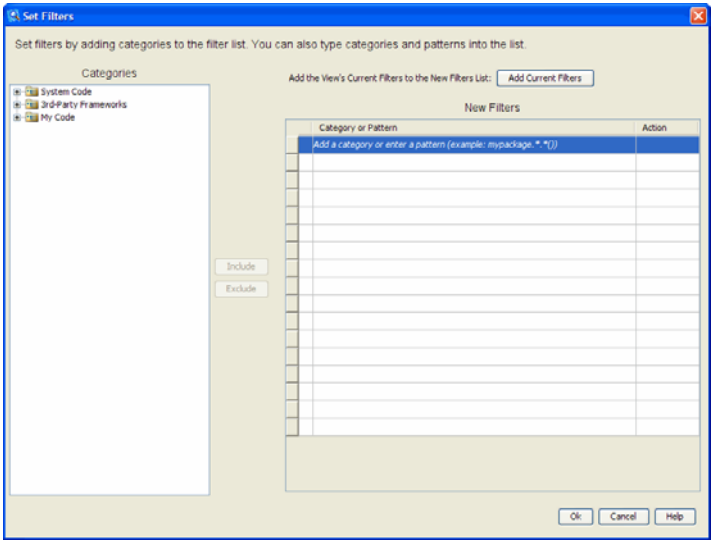
   The table refreshes, listing only the classes that meet the criteria you entered. The number of classes/total classes is displayed beside the text field. Note that the chart is not affected.

Note The filters you enter in this field are saved and can be used later by clicking the down arrow and choosing them from the drop-down list.

To filter the Instances view using the Set Filters dialog:

1. Click **Set Filters** .

   The *Set Filters* dialog box opens.



2 Add filters to the New Filters list in one or more of the following ways:

- Click a line in the list to highlight it and type a pattern.
- Click in a line in the list to highlight it, then click the browse button to navigate to a class or method, and double-click it.
- Click **Add Current Filters** to add the filters that are currently in effect in the Instances view.
- Navigate to a class or method in the Categories pane and click **Include**.

3 If you want to change the action for a filter, click in the **Action** column and choose the **Include** or **Exclude**.

4 Click **OK** when you have finished adding filters.

The view refreshes and displays only the instances that meet the filter criteria you have set. The bar chart is redrawn to reflect the matched instances.

Syntax for Instance Filters

Filters are case-sensitive and blank spaces are not allowed. You can use an asterisk (*) as a wild card; it matches any character, including the package separator. If you specify the code in a shorthand or partial form, JProbe automatically changes it to canonical form. The change may not be exactly what you desire, so be sure to verify it. JProbe does not check to see whether the filter actually matches anything in your code. For

example, if you make a typing error in your package name, nothing in your code will match the filter and so the filter is not effective.

The following table contains common syntax and describes how JProbe interprets it:

If you type this:	Canonical form is this:	And filters are applied to:
`*`	`*.*.*()`	All methods in all classes in all packages (including the unnamed package)
`.*`	`.*.*()`	All methods in all classes in the unnamed package only
`C`	`*.C.*()`	All methods in classes named `C` in any package
`.C`	`.C.*()`	All methods in the class named `C` in the unnamed package only
`M()`	`*.*.M()`	Method `M` in all classes in all packages
`P.String*`	`P.String*.*()`	All methods in any class in the package `P` whose name starts with `String` and all methods in any class in a package or subpackage of `P.String*` (matches both `P.StringArt.draw()` and `P.Stringent.Check.English.spelling()`)
`P.*.C.do*()`	*no change*	Any method whose name starts with `do` in any class named `C` in any subpackage of `P`, except `P` directly (matches `P.SubPkg.C.doGet()`, but does not match `P.C.doNotMatch()`)
`Pre*`	`*.Pre*.*()`	All methods in all classes starting with `Pre` in any package, as well as any class in a subpackage whose name starts with `Pre` (matches both `Pkg.Prepare.m()` and `Pkg.Pretend.C.m()`)

If you type this:	Canonical form is this:	And filters are applied to:
`s*y`	`*.s*y.*()`	All methods in classes whose name ends with `y` and either starts with `s` or belongs to a subpackage that starts with `s` (matches both `java.rmi.server.RMISocketFactory`, `com.quest.say`, and `java.security.cert.X509CRLEntry`)
`foo.bar`	`foo.bar.*()`	All methods in the class `bar` in package `foo`. This is likely not what you wanted. See the next example.
`foo.bar.*`	`foo.bar.*.*()`	All methods in all classes in `foo.bar` or any of its subpackages
`foo.bar.String.t*e()`	*no change*	Methods whose names start with `t` and end in `e` in `foo.bar.String` (matches both `toUpperCase()` and `toLowerCase()`)

Interacting with the Instances View

The following table summarizes the tasks that you can perform:

Task	Action	Response
See the instances that dominate the selection	Select a class from the table and click **Dominating Instances**  . OR Select a class from the table, right-click, and select the **Why Are These Live?** option from the list.	The Dominating Instances view opens, showing what instances are dominating (pinning) the selected instance set. For more information, see “Exploring the Dominating Instances View” on page 148.

Task	Action	Response
See the instances that are pinned by the selection	Select a class from the table and click **Pinned Instances**  . OR Select a class from the table, right-click, and select the **What Do These Pin?** option from the list.	The Pinned Instances view opens, showing the instances dominated (pinned by) the selected instance set. For more information, see "Exploring the Pinned Instances View" on page 155.
See the instances for a class or classes	Select a class from the table and click **Instance Detail View**  . **Note** This action is available only if the selected classes have instances that are live in the heap.	The Instance Detail view opens with all the instances for the selected class listed in the upper pane. For more information, see "Exploring the Instance Detail View" on page 166.
View the merged allocation points for the instances of one or more classes	Set Investigate By to **Recorded Count/Memory** or **Dead Count/Memory**. Select a class from the table and click **Merged Allocation Points**  . **Note** This action is available only if traces were collected for the selected classes.	The Merged Allocation Points view opens with the merged stack traces for the selected classes. For more information, see "Exploring the Merged Allocation Points View" on page 181.
See a call tree/graph for selected instances or garbage data	Set Investigate By to **Recorded Count/Memory** or **Dead Count/Memory**. Select a class from the table and click **Call Traces**  . **Note** This action is available only if traces were collected for the selected classes.	The Call Traces view opens, showing the allocation call tree and graph for the selected instance(s). For more information, see "Exploring the Call Traces View" on page 174.

Task	Action	Response
Show allocations of large number of objects	Select a class from the table and click **Show Allocations View**  . **Note** This action is available only if trace information was collected.	The Allocations view opens, showing all the methods that allocated instances. For more information, see "Exploring the Allocations View" on page 160.
Drill down into the data	Click a segment of the bar chart.	The Instances view is redrawn to display only the instances in the chosen category.
Filter the table	See "Filtering Data in the Instances View" on page 141.	The classes in the table reflect the filtering expression(s) you have entered.
Calculate Actual Keep Alive Size	The ~ sign in the Keep Alive Size column indicates an estimated value for this metric. To calculate the actual size, select a row, right-click, and select **Calculate Actual Keep Alive Size** from the list.	The Keep Alive Size column is updated with the actual Keep Alive Size value.
Set your primary investigative path	Select one of the following options from the *Investigate By* list: • To investigate recorded objects, select **Recorded Count/ Memory**. • To investigate the whole heap, select **Heap Count**/ **Memory**. • To investigate garbage data, select **Dead Count**/**Memory**.	The horizontal bar graph is re-drawn to reflect your choice.

Task	Action	Response
Save the chart as a PNG or JPEG file	Right-click in the chart and select **Save Chart As**, then select the graphic file type (**PNG** or **JPEG**) in which you want to save the chart. In the *Export Image* dialog box, specify the export parameters and a location to save the file.	The file is saved to the location you indicate.
Export the chart to CSV	Right-click in the chart and select **Export to CSV**. In the *Export to CSV* dialog box, specify the export parameters and a location to save the file.	The file is saved to the location you indicate. The file contains the table data separated by a comma. It may also contain headers. For more information, see "Changing How JProbe Exports to a CSV File" on page 455.
Show/hide table columns	Right-click the table and select **Show/Hide Columns**. In the *Show/Hide Columns* dialog box, select the columns to show and click **OK**.	The selected columns are displayed. The rest are hidden.
Resize column widths	To set the column width, drag the edge of the column heading left or right. To see all data in a column, right-click in the column and select **Size <column name> to Fit**. To fit all columns in the viewing area, right-click and select **Size All Columns to Fit**.	The columns are resized.
Rearrange columns	Drag a column by its heading and drop it in a new location.	The column appears in its new location.

Task	Action	Response
Sort the table by a metric	Click the column heading for the metric.	The table is sorted. To reverse the sort order, click the heading again.
Export the table to CSV	Right-click in the table and select **Export to CSV**. Type a file name and click **Save**.	The file contains the table data separated by a comma. It may also contain headers. For more information, see "Changing How JProbe Exports to a CSV File" on page 455.
Create a report for this view	Click **Export Data to HTML**  .	The *Create Instances Report* dialog box appears. For more information, see "Instances Report" on page 204.

Exploring the Dominating Instances View

This section contains the following information:

- "Opening the Dominating Instances View" on page 148
- "Understanding the Dominating Instances View" on page 149
- "Filtering Data in the Dominating Instances View" on page 151
- "Interacting with the Dominating Instances View" on page 152

Opening the Dominating Instances View

The Dominating Instances view can be reached only from the Instances view.

To open the Dominating Instances view:

1 Select a class in the *Instances* view.

2 Click **Dominating Instances**  .

or

Right-click the selected row and select the **Why Are These Live?** option from the list.

The *Dominating Instances* view appears.

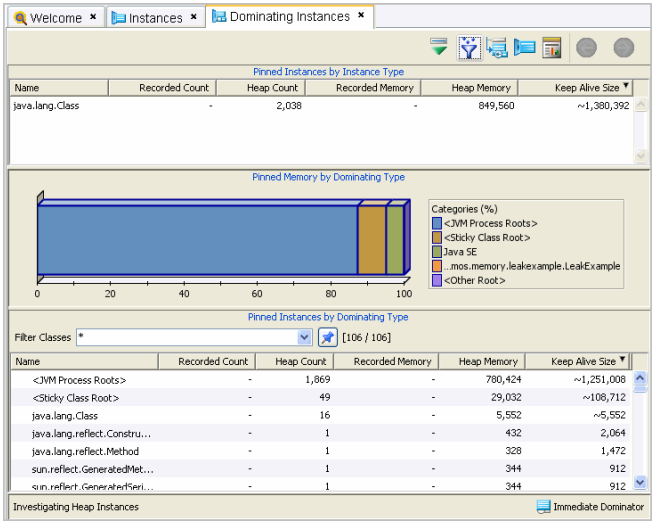
Understanding the Dominating Instances View

The Dominating Instances view allows you to view the instances (referred to as dominators) that are holding a specific set of instances in the heap and preventing them from being garbage collected.

Dominators offer a different way of viewing heap data, based on reachability analysis. If every path from roots in the heap must pass through one instance (for example, a Widget) before getting to another (for example, a String), the former is said to dominate the latter (the Widget dominates the String).

During snapshot processing, JProbe analyzes the paths in the heap and identifies which instances are reachable only through other instances. It builds a tree structure, called a dominator tree, which starts with the roots of the heap and includes all instances in the heap. Using the Dominating Instances view, you can analyze “why an instance is live”, by navigating towards the roots of the tree.

The Dominating Instances view consists of three panes.



The upper pane, *Pinned Instances by Instance Type*, contains a table which shows details about the instance set selected in the *Instances* view. Each row represents the set of pinned instances that you selected as input to the view. This is the “starting set” of instances from which the data in the rest of the view is derived. These instances are grouped by class, similar to the *Instances* view table.

The middle pane, *Pinned Memory by Dominating Type*, shows the categories of instances responsible for pinning memory in the “starting set”. The pane contains a chart which illustrates the percentage of dominating instances by Keep Alive Size in each of the categories that are listed in the legend to the right of the chart. Hovering over a segment of the chart displays a popup showing the memory kept alive by that category. Selecting a bar in the chart redraws the Dominating Instances view using instances from the selected category as the “starting set”. This allows the investigation of “why instances are live” to continue towards the roots that pin the Java heap.

The lower pane, *Pinned Instances by Dominating Type*, contains a table which shows the breakdown of classes responsible for pinning the instances listed in the upper pane. The values displayed in the Recorded Count, Heap Count, Recorded Memory, Heap Memory, and Keep Alive Size columns in the lower pane represent a subset of the total shown in the corresponding columns in the upper pane. For example, by summing up

the Heap Count rows in the lower pane you get a value that corresponds to the Heap Count values in the upper pane.

Note The Keep Alive Size values are estimates, so the sum of the values in the lower pane differs from those in the upper pane.

The status bar displays the most common icons and short descriptions of what they represent.

In the *Dominating Instances* view, you can look at the lower pane and quickly see that a number of the instances are pinned by a particular class. The middle pane offers a breakdown of this information using categories (so you can see which category of classes are holding on to the most of your "starting set" of instances). A good approach is to search for the highest count or keep-alive size and focus on those instances (by asking **Why Are These Live?** to repeat the process, or by selecting the appropriate bar from the chart in the middle pane). Once you have reduced the set of instances to a few that seem to be responsible for pinning the majority of memory, it is worth opening the *Instance Detail* view.

You can drill down in a category by clicking either a section in the chart or the corresponding category in the legend. As you drill down to more detailed information, the chart and table refresh to show only the instances in that category. You can click the back and forward toolbar buttons to move backward and forward through operations you have already performed, or you can click the down arrow beside either button and go directly to any of the other operations that you have performed (as shown below).

Similar to the Instances view, you can sort the tables by any of the columns. The text field above the table allows you to filter the classes that are listed in the table. See "Filtering Data in the Instances View" on page 141 for details.

Filtering Data in the Dominating Instances View

In the Dominating Instances view, the filters set under the **Set Preferred Dominators** icon are used to determine which dominators are worth investigating during the memory analysis.

In some cases, you may only be interested in the instances that your code is pinning. If filters are set to include your code, the dominators for each instance are searched for instances that match that filter. If an instance has a dominator that matches the filter, that dominator's class is considered a "preferred dominator" and presented in favour of the

instance's immediate dominator in the table (in the lower pane), as well as in the middle pane's chart. If the filter was not matched, the instance's immediate dominator is used instead.

Interacting with the Dominating Instances View

The following table summarizes the tasks that you can perform:

Task	Action	Response
See the instances pinned by the selected dominators	Select a class and click **Instances Detail View**  . OR Select a class, right-click, and select the **Show This Dominator's Pinned Instances** option from the list. **Note** This action is available only if the selected classes have instances that are live in the heap.	The Instance Detail view opens with all the instances for the selected class listed in the upper pane.
Focus on instances pinned by your code	Click **Set Preferred Dominators**  .	The *Set Filters* dialog box opens. The classes in the table reflect the filtering expression(s) you have entered. For more information, see "Filtering Data in the Dominating Instances View" on page 151.
Show immediate dominators	Click **Show All Immediate Dominators**  .	The lower pane is updated to display all immediate dominator instances.

Show preferred dominators	Click **Show Preferred Dominators**  .	The lower pane is updated to display only the dominating types defined as “preferred” through the use of filters. If a preferred dominator is not found, its immediate dominator is shown.
See the instances that dominate a set of instances	Select a class, right-click, and select the **Why Are These Live?** option from the list.	The Dominating Instances view opens, showing what instances are dominating (pinning) the selected instance set.
See the instances pinned by the selected dominator	Select a class, right-click, and select the **Show This Dominator’s Pinned Instances** option from the list.	The Instance Detail view opens and displays the instances pinned by the selected dominating type.
Calculate Actual Keep Alive Size	The ~ sign in the Keep Alive Size column indicates an estimated value for this metric. To calculate the actual size, right-click the row and select **Calculate Actual Keep Alive Size** from the list.	The Keep Alive Size column is updated with the actual Keep Alive Size value.
Save the chart as a PNG or JPEG file	Right-click in the chart and select **Save Chart As**, then select the graphic file type (**PNG** or **JPEG**) in which you want to save the chart. In the *Export Image* dialog box, specify the export parameters and a location to save the file.	The file is saved to the location you indicate.

Show/hide table columns	Right-click the table and select **Show/Hide Columns**. In the *Show/Hide Columns* dialog box, select the columns to show and click **OK**.	The selected columns are displayed. The rest are hidden.
Resize column widths	To set the column width, drag the edge of the column heading left or right. To see all data in a column, right-click in the column and select **Size <column name> to Fit**. To fit all columns in the viewing area, right-click and select **Size All Columns to Fit**.	The affected columns are resized.
Rearrange columns	Drag a column by its heading and drop it in a new location.	The column appears in its new location.
Sort the table by a metric	Click the column heading for the metric.	The table is sorted. To reverse the sort order, click the heading again.
Export table data to CSV	Right-click in the table and select **Export to CSV**. Type a file name and click **Save**.	The file contains the table data separated by a comma. It may also contain headers. For more information, see "Changing How JProbe Exports to a CSV File" on page 455.
Create a report for this view	Click **Export Data to HTML**  .	The *Create Dominating Instances Report* dialog box appears. For more information, see "Dominating Instances Report" on page 205.

Exploring the Pinned Instances View

This section contains the following information:

- "Opening the Pinned Instances View" on page 155
- "Understanding the Pinned Instances View" on page 155
- "Interacting with the Pinned Instances View" on page 157

Opening the Pinned Instances View

The Pinned Instances view can be reached only from the Instances view.

To open the Pinned Instances view:

1. Select a class in the Instances view.
2. Click **Pinned Instances**  .

   or

   Right-click the selected class and select the **What Do These Pin?** option from the list.

   The *Pinned Instances* view appears.

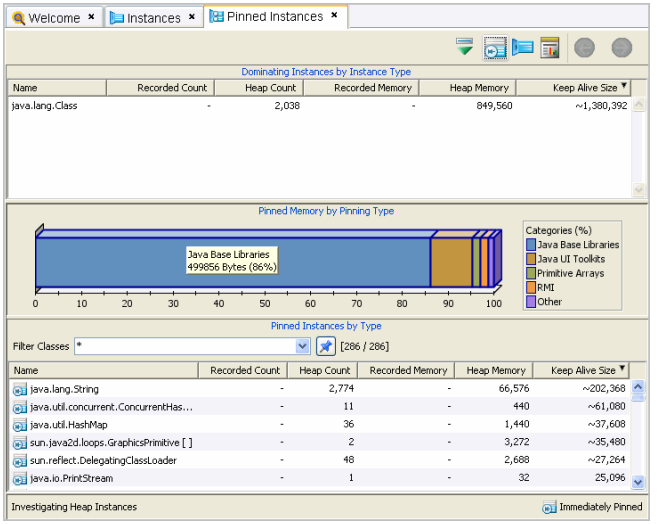
Understanding the Pinned Instances View

The Pinned Instances view allows you to view the instances that are being held in the heap by a specific set of instances (referred to as dominators) and, as a consequence, cannot be garbage collected.

Dominators offer a different way of viewing heap data, based on reachability analysis. If every path from roots in the heap must pass through one instance (for example, a Widget) before getting to another (for example, a String), the former is said to dominate the latter (the Widget dominates the String).

During snapshot processing, JProbe analyzes the paths in the heap and identifies which instances are reachable only through other instances. It builds a tree structure, called a dominator tree, which starts with the roots of the heap and includes all instances in the heap. Using the Pinned Instances view, you can analyze "what's pinning an instance", by navigating towards the leaves of the tree.

The Pinned Instances view consists of three panes.



The upper pane, *Dominating Instances by Instance Type*, contains a table which shows details about the instance set selected in the *Instances* view. Each row represents the set of dominating instances that you selected as input to the view (when selecting **What Do These Pin?** option from the list) and they are responsible for keeping alive the instances shown in the middle and lower panes. The upper pane shows the “starting set” of instances from which the data in the rest of the view is derived. These instances are grouped by class, so instead of seeing a row for each instance, you can see only one row summarizing the count of instances (similar to the *Instances* view table).

The middle pane, *Pinned Memory by Pinning Type*, shows the categories of instances pinned by the “starting set”. The pane contains a chart which illustrates the percentage of pinned instances by Keep Alive Size in each of the categories that are listed in the legend to the right of the chart. Hovering over a segment of the chart displays a popup showing the memory used by that category. Selecting a bar from the chart redraws the Pinned Instances view using the instances from the category chosen as the “starting set”.

The lower pane, *Pinned Instances by Type*, contains a table which shows the instances pinned by the “starting set” of instances listed in the upper pane. For example, if you go into the view with 30 `HashMaps` instances as the starting set, you may see that they pin `HashMap$Entry[]`, `HashMap$Entry`, and (often) `Strings`. The values displayed in the Recorded Count, Heap Count, Recorded Memory, Heap Memory, and Keep Alive Size columns in the lower pane are for the number of `HashMap$Entry[]/ HashMap$Entry/ String` instances that are pinned. The middle pane summarizes the largest rows based on the Keep Alive Size value.

The status bar displays the most common icons and short descriptions of what they represent.

In the *Pinned Instances* view, you can navigate towards the leaves of the dominator tree. The "starting set" shown in the upper pane pins a number of instances, those instances are grouped based on categories (in the middle pane) and shown in detail in the lower pane.

You can drill down in a category by clicking either a section in the chart or the corresponding category in the legend. As you drill down to more detailed information, the chart and table refresh the view using only the instances in that category as the "starting set". You can click the back and forward toolbar buttons to move backward and forward through operations you have already performed, or you can click the down arrow beside either button and go directly to any of the other operations that you have performed (as shown below).

Similar to the Instances view, you can sort the tables by any of the columns. The text field above the table allows you to filter the classes that are listed in the table. See "Filtering Data in the Instances View" on page 141 for details.

Interacting with the Pinned Instances View

The following table summarizes the tasks that you can perform:

Task	Action	Response
See the instances pinned by the selected dominators	Select a class and click **Instance Detail View**. OR Select a class, right-click, and select the **Show These Pinned Instances** option from the list. **Note** This action is available only if the selected classes have instances that are live in the heap.	The Instance Detail view opens with all the instances for the selected class listed in the upper pane.

Show immediately pinned instances	Click **Show Immediately Pinned Instances**  .	The lower pane is updated to display only instances immediately pinned by the "starting set".
Show all pinned instances	Click **Show All Pinned Instances**  .	The lower pane is updated to display all instances pinned by the "starting set", whether immediately pinned or pinned indirectly through another instance.
See the instances that are pinned by a dominating instance	Select a class, right-click, and select the **What Do These Pin?** option from the list.	The Pinned Instances view is updated to display the instances dominated (pinned by) the selected instance set.
Calculate Actual Keep Alive Size	The ~ sign in the Keep Alive Size column indicates an estimated value for this metric. To calculate the actual size, right-click the row and select **Calculate Actual Keep Alive Size** from the list.	The Keep Alive Size column is updated with the actual Keep Alive Size value.
Save the chart as a PNG or JPEG file	Right-click in the chart and select **Save Chart As**, then select the graphic file type (**PNG** or **JPEG**) in which you want to save the chart. In the *Export Image* dialog box, specify the export parameters and a location to save the file.	The file is saved to the location you indicate.
Show/hide table columns	Right-click the table and select **Show/Hide Columns**. In the *Show/Hide Columns* dialog box, select the columns to show and click **OK**.	The selected columns are displayed. The rest are hidden.

Resize column widths	To set the column width, drag the edge of the column heading left or right. To see all data in a column, right-click in the column and select **Size <column name> to Fit**. To fit all columns in the viewing area, right-click and select **Size All Columns to Fit**.	The columns are resized.
Rearrange columns	Drag a column by its heading and drop it in a new location.	The column appears in its new location.
Sort the table by a metric	Click the column heading for that metric. To reverse the sort order, click the heading again.	The table is sorted.
Export table data to CSV	Right-click in the table and select **Export to CSV**. Type a file name and click **Save**.	The file contains the table data separated by a comma. It may also contain headers. For more information, see “Changing How JProbe Exports to a CSV File” on page 455.
Create a report for this view	Click **Export Data to HTML**  .	The *Create Pinned Instances Report* dialog box appears. For more information, see “Pinned Instances Report” on page 205.

Exploring the Allocations View

This section contains the following information:

- "Opening the Allocations View" on page 160
- "Understanding the Allocations View" on page 160
- "Interacting with the Allocations View" on page 161

Opening the Allocations View

The Allocations view can be reached from the Instances view, only if trace information was collected for the selected classes.

To open the Allocations view:

- In the Instances view, click **Show Allocations View**  .

  The Allocations view appears.

Understanding the Allocations View

When analyzing your application's memory usage, it is beneficial to view data from several different perspectives. If you noticed steep increases in the heap usage at certain points, a method might be allocating a large number of objects. The Allocations view offers you a different way of viewing object allocations. This can help you to determine if a method is over-allocating objects in your code.

By default, the view displays a list of methods that have allocated live instances during the use case. Once you have established which method is over-allocating objects, you can further your investigation by viewing related objects using the Location Detail view.

The Allocations view provides you with multiple ways of viewing the same method data. The *Group Data By* list provide you with a choice of how you want to view data. It offers the following options: by Package, Class, and Method.

Interacting with the Allocations View

The following table summarizes the tasks that you can perform:

Task	Action	Response
Investigate objects allocated by the selected package, class, or method	Select a package, class, or method and click **Show Allocations Detail**  . OR Select a class, right-click, and select the **Show Allocations Detail** option from the list.	The Allocations Detail view opens and displays all objects allocated directly or indirectly by the selected package, class, or method. For more information, see "Exploring the Allocations Detail View" on page 163. **Note** If filters are used in the Allocation view, the list contains objects that match the display filter.
View the source code for the selected allocation	Select a class and click **Show Allocated At Source**  . OR Select a class, right-click, and select the **Show Allocated At Source** option from the list. **Note** If this menu item is disabled, JProbe cannot associate a source file with the class file.	A browser opens and you can search for the source file of your application. The source file opens at the code line defining the selected allocation. For more information, see "Exploring the Allocations Detail View" on page 163.
Select how to display the allocation data	Select one of the available options from the *Group Data By* list.	The table is updated to reflect the selected option.

Filter the table	Type a filtering expression in the Filter Allocation Points text box, then press **Enter**.	The classes in the table reflect the filtering expression(s) you have entered. For more information, see “Filtering Data in the Instances View” on page 141.
Show/hide table columns	Right-click the table and select **Show/Hide Columns**. In the *Show/Hide Columns* dialog box, select the columns to show and click **OK**.	The selected columns are displayed. The rest are hidden.
Resize column widths	To set the column width, drag the edge of the column heading left or right. To see all data in a column, right-click in the column and select **Size <column name> to Fit**. To fit all columns in the viewing area, right-click and select **Size All Columns to Fit**.	The columns are resized.
Rearrange columns	Drag a column by its heading and drop it in a new location.	The column appears in its new location.
Sort the table by a metric	Click the column heading for that metric. To reverse the sort order, click the heading again.	The table is sorted.
Export table data to CSV	Right-click in the table and select **Export to CSV**. Type a file name and click **Save**.	The file contains the table data separated by a comma. It may also contain headers. For more information, see “Changing How JProbe Exports to a CSV File” on page 455.

Create a report for this view	Click **Export Data to HTML**  .	The *Create Allocations Report* dialog box appears. For more information, see "Allocations Report" on page 206.

Exploring the Allocations Detail View

This section contains the following information:

- "Opening the Allocations Detail View" on page 163
- "Understanding the Allocations Detail View" on page 163
- "Interacting with the Allocations Detail View" on page 164

Opening the Allocations Detail View

The Allocations Detail view can be reached from the Allocations view.

To open the Allocations Detail view:

1. Select a package, class, or method in the Allocations view.
2. Click **Show Allocations Detail**  .

   The Allocations Detail view appears.

Understanding the Allocations Detail View

The Allocations Detail view displays a list of objects that were allocated by the selected item (package, class, or method) in the Allocations view and provides in depth information about that particular allocation.

You can view data that is allocated directly and data that is allocated indirectly. By default, this view displays cumulative data. Objects allocated directly and indirectly are included in the list. Click **Show Directly Allocated Instances**  to include only those objects that were allocated directly. Click **Show All Allocated Instances**  to return to the list of objects allocated directly and indirectly.

If you suspect that one of these objects may be causing a problem in your code, you can use the Instance Detail view or Merged Allocation Point view to continue your investigation of the object.

Interacting with the Allocations Detail View

The following table summarizes the tasks that you can perform:

Task	Action	Response
View only directly allocated objects	Click **Show Directly Allocated Instances**  .	The table is updated to reflect the selected option.
View all allocated objects (directly and indirectly)	Click **Show All Allocated Instances**  .	The table is updated to reflect the selected option.
Examine the instances for the allocated object	Select a class and click **Instance Detail View**  . OR Select a class, right-click, and select the **Open Instance Detail View** option from the list.	The Instance Detail view opens with all the instances for the selected class listed in the upper pane. For more information, see "Exploring the Instance Detail View" on page 166.
Examine merged stack traces	Select a class and click **Merged Allocation Points View**  . OR Select a class, right-click, and select the **Open Merged Allocation Points View** option from the list.	The Merged Allocation Points view opens and displays the merged stack trace for the class selected from the Allocation Detail view. To follow a stack trace, click the plus sign to expand the list. For more information, see "Exploring the Merged Allocation Points View" on page 181.

Examine the allocation call tree and graph	Select a class and click **Call Traces View**  . OR Select a class, right-click, and select the **Open Call Traces View** option from the list.	The Call Traces view opens displaying the call tree and graph for the selected instances. For more information, see "Exploring the Call Traces View" on page 174.
Show/hide table columns	Right-click the table and select **Show/Hide Columns**. In the *Show/Hide Columns* dialog box, select the columns to show and click **OK**.	The selected columns are displayed. The rest are hidden.
Resize column widths	To set the column width, drag the edge of the column heading left or right. To see all data in a column, right-click in the column and select **Size <column name> to Fit**. To fit all columns in the viewing area, right-click and select **Size All Columns to Fit**.	The columns are resized.
Rearrange columns	Drag a column by its heading and drop it in a new location.	The column appears in its new location.
Sort the table by a metric	Click the column heading for that metric. To reverse the sort order, click the heading again.	The table is sorted.
Export table data to CSV	Right-click in the table and select **Export to CSV**. Type a file name and click **Save**.	The file contains the table data separated by a comma. It may also contain headers. For more information, see "Changing How JProbe Exports to a CSV File" on page 455.

Create a report for this view	Click **Export Data to HTML**  .	The *Create Allocations Detail Report* dialog box appears. For more information, see "Allocations Detail Report" on page 206.

Exploring the Instance Detail View

The Instance Detail view allows you to extensively analyze the instances of a class. You can see the instance's reference and referrer trees, as well as a class list of dominating instances.

This section contains the following information:

- "Opening the Instance Detail View" on page 166
- "Understanding the Instance Detail View" on page 167
- "Interacting with the Instance Detail View" on page 171

Opening the Instance Detail View

You can drill down to more detail on the instances of a class from several different views. The Instance Detail view is reachable from most Memory views.

To open the Instance Detail view:

1. Select a row and click **Instance Detail View**  .

   The Instance Detail view opens. The instances for the selected class or method are displayed in the upper pane.

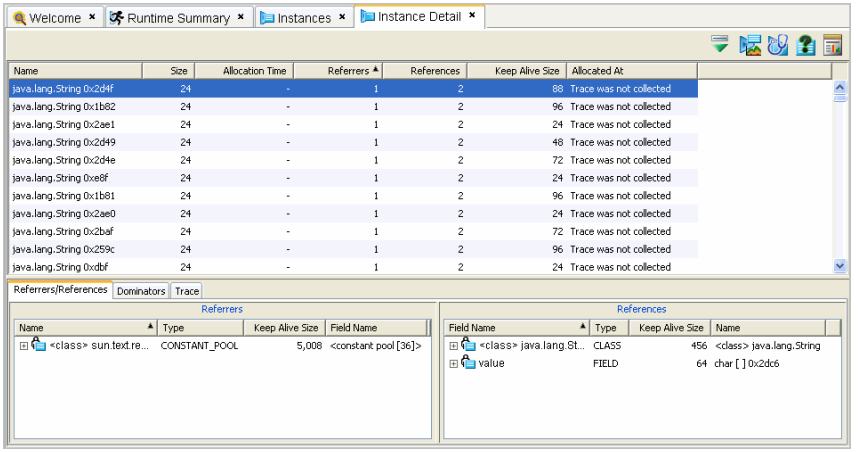
To open the Instance Detail view with a single instance:

1. Select an instance in the Heap Graph.
2. Do one of the following:
   - Double-click the instance.
   - Click **Instance Detail View**  .

   The Instance Detail view opens with the selected instance in the instance list.

Understanding the Instance Detail View

The Instance Detail view has two panes. The upper pane displays a list of instances collected during data recording. The bottom pane has three tabs. One displays the referrers and references for the selected instance; the second displays the dominators of the selected instance, and the third displays the allocation stack trace for the selected instance.


Welcome
Runtime Summary
Instances
Instance Detail
Name
Size
Allocation Time
Referrers
References
Keep Alive Size
Allocated At
java.lang.String 0x2d4f 24 - 1 2 88 Trace was not collected
java.lang.String 0x1b82 24 - 1 2 96 Trace was not collected
java.lang.String 0x2ae1 24 - 1 2 24 Trace was not collected
java.lang.String 0x2d49 24 - 1 2 48 Trace was not collected
java.lang.String 0x2d4e 24 - 1 2 72 Trace was not collected
java.lang.String 0xe8f 24 - 1 2 24 Trace was not collected
java.lang.String 0x1b81 24 - 1 2 96 Trace was not collected
java.lang.String 0x2ae0 24 - 1 2 24 Trace was not collected
java.lang.String 0x2baf 24 - 1 2 72 Trace was not collected
java.lang.String 0x259c 24 - 1 2 96 Trace was not collected
java.lang.String 0xdbf 24 - 1 2 24 Trace was not collected
Referrers/References
Dominators
Trace
Referrers
Name
Type
Keep Alive Size
Field Name
<class> sun.text.re... CONSTANT_POOL 5,008 <constant pool [36]>
References
Field Name
Type
Keep Alive Size
Name
<class> java.lang.St... CLASS 456 <class> java.lang.String
value FIELD 64 char [ ] 0x2dc6

Instance List

The upper pane contains the following data about each occurrence of an instance:

Name — Name and memory address.

Size — Memory used by the object.

Allocation Time — The time when the object was created.

> **Note** The Allocation Time is only displayed for instances for which allocation data was recorded. If allocation data was not recorded, an allocation time of 00:00:000 is displayed.

Referrers — The number of referrers to the instance.

References — The number of references to other instances.

Keep Alive Size	The amount of memory that would be released if the object is removed from the heap.
Allocated At	The method that allocated the instance or the package for the class containing the allocating method.

Referrers/References

The instance selected in the instance list forms the root of the Referrers tree. By expanding the branches, you can discover the path of references that attaches this instance to a live object and, ultimately, to the root set.

Loitering objects are caused by a live object inadvertently or unnecessarily retaining a reference to the loiterer for some period of time. To allow the garbage collector to remove a loitering object and all its descendants you must remove all references to the loiterer. When you locate your loiterer, you can look at the References pane to see how many objects a loiterer anchors.

The Referrers tree provides the following information:

Name	The instance that is referring to the instance that you are investigating.
Type	The type of the reference.
Keep Alive Size	The amount of memory that would be released if the object is removed from the heap.
Field Name	The field in the referring instance that refers to the referenced instance.

The References tree provides the following information:

Field Name	The field in the referring instance that refers to the referenced instance.
Type	The type of the reference.
Keep Alive Size	The amount of memory that would be released if the object is removed from the heap.

Name	The name of the field that references the instance in the tree.

The name of each field in the Referrers/References trees is prefixed by an icon which provides information about that field, as described in the following table.

Icon	Name	Type
	Recorded	Instance created when the recording was enabled (On).
	Not Recorded	Instance created when the recording was disabled (Off).
	Root	Root instance.
	Static Root	Static root instance.
	Cycle	Instance that refers to itself (circular reference).

Note The status bar also displays the most common icons and short descriptions of what they represent.

Dominators

The **Dominators** tab contains an expanded tree-table that contains a chain of referrers back to a root, highlighting dominating instances (). The table provides the following information:

Name	The instance that is referring to the instance that you are investigating.
Type	The type of the reference.
Field Name	The field in the referring instance that refers to the referenced instance.

Size	Memory used by the object.
Allocation Time	The time when the object was created.
Keep Alive Size	The amount of memory that would be released if the object is removed from the heap.

Traces

The **Allocation Stack Trace** tab contains information about the instance selected in the instance list. If class file contained line and source information, the class name and line number for the allocating code appear in the Source column.

Note In order for Traces to be available, you must have collected trace information during data recording.

The Trace table contains the following data:

Method	Class and method names for the methods that allocated instances of the selected class.
Source	The name of the class file and the line number of the method.

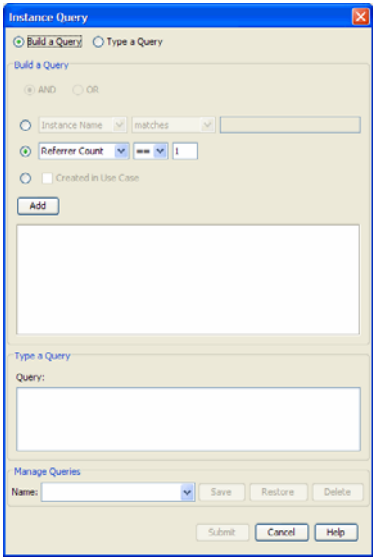
Querying a Set of Instances

If there is a large number of instances in the view, you can use a query to narrow the set of instances that you are investigating.

To narrow a set of instances using a query:

1. Select an instance in the Instance Detail view.
2. Click **Query**  .

   The *Instance Query* dialog box opens.



3 Build a query using the options provided, and click **Add** to display them in the text field.

   or

   Choose **Type a Query** and type your query in the **Query** field.

4 When you have built or typed your query, you can save it by giving it a name and clicking **Save**.

5 Click **Submit** to execute the query.

   All the instances that do not meet the parameters of the query are removed from the list.

Interacting with the Instance Detail View

You can select a new instance and review its tree of referrers or references. In the tree, you can expand and collapse branches. At any time, you can view source code for a class from an instance or allocating method.

The following table summarizes the tasks that you can perform:

Task	Action	Response
See Referrer/ Reference data for an instance	In the instance list, click an instance row.	The **Referrers/References** tab displays data for the instance.
See Dominators for an instance	In the instance list, click an instance row.	The **Dominators** tab displays data for the instance.
Expand or collapse a tree or branch	Double-click the **Referrer** or **Reference** instance (or click the expand symbol).	If collapsed, the branch is expanded. If expanded, the branch is collapsed.
View the source code for an allocating method	Select an instance, right-click, and select **Show Allocated At Source**. **Note** If this menu item is disabled, JProbe cannot associate a source file with the class file.	If you have the source code for the class containing this method, the Source view opens with the code displayed.
See the instances that dominate a set of instances	Select a class, right-click, and select the **Why Are These Live?** option from the list.	The Dominating Instances view opens, showing what instances are dominating (pinning) the selected instance set.
See the instances that are pinned by a dominating instance	Select a class, right-click, and select the **What Do These Pin?** option from the list.	The Pinned Instances view is updated to display the instances dominated (pinned by) the selected instance set.
Narrow the set of instances in the view	Select one or more classes and click **Query**  .	The *Query* dialog box opens.

Task	Action	Response
Examine an instance and its referrers and references as a directed graph	Select an instance and click **Heap Graph**  .	The Heap Graph view opens with the instance selected. In the graph, only the instance, its referrers, and its references are displayed as nodes.
Investigate an instance using the Leak Doctor	Select an instance and click **Leak Doctor**  .	The Leak Doctor view opens with the instance selected.
Show/hide table columns	Right-click the table and select **Show/Hide Columns**. In the *Show/Hide Columns* dialog box, select the columns to show and click **OK**.	The selected columns are displayed. The rest are hidden.
Resize column widths	To set the column width, drag the edge of the column heading left or right. To see all data in a column, right-click in the column and select **Size <column name> to Fit**. To fit all columns in the viewing area, right-click and select **Size All Columns to Fit**.	The columns are resized.
Rearrange columns	Drag a column by its heading and drop it in a new location.	The column appears in its new location.
Sort the table by a metric	Click the column heading for that metric. To reverse the sort order, click the heading again.	The table is sorted.

Task	Action	Response
Export the instance table data to CSV	Right-click in the table and select **Export to CSV**. Type a file name and click **Save**.	The file contains the table data separated by a comma. It may also contain headers. For more information, see "Changing How JProbe Exports to a CSV File" on page 455.
Create a report for this view	Click **Export Data to HTML**  .	The *Create Instance Detail Report* dialog box appears. For details, see "Instance Detail Report" on page 207.

Exploring the Call Traces View

When analyzing your application's memory usage, it is helpful to view data from several different perspectives. If you noticed steep increases in the heap usage at certain points, a method might be allocating a large number of objects. The Call Traces view offers you a different way of looking at object allocations. This can help you to determine if a method is over-allocating objects in your code.

Note This view is not available if you have set the Investigate By option to Heap Count or Heap Memory in the Instances view. For more information, see "Understanding the Instances View" on page 139..

When you are investigating by Recorded Count or Memory, the view displays a list of methods that have allocated instances during data recording and were still live when the snapshot was taken.

Note In order for the Call Traces view to be available, you must have collected trace information during data recording.

This section contains the following information:

- "Opening the Call Traces View" on page 175
- "Understanding the Call Traces View" on page 175

- "Customizing the Call Traces View" on page 177
- "Interacting with the Call Traces View" on page 178

Opening the Call Traces View

If you took a snapshot with stack traces during a session, you can view recorded and garbage collection data in the Call Traces view.

To open the Call Traces view:

1. In the Snapshots Navigator, double-click the snapshot you want to display.

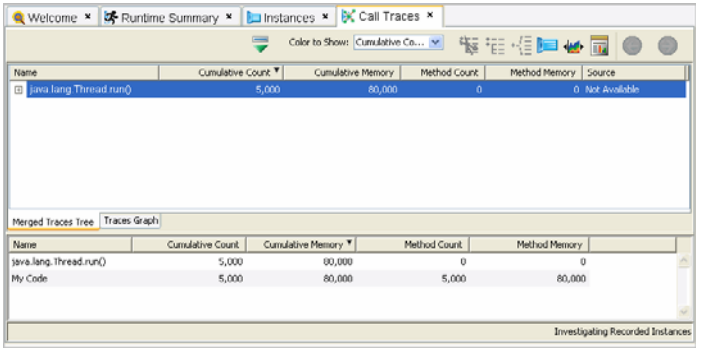
   The Instances view displays the object data contained in the snapshot.

2. From the *Investigate by* list, select **Recorded Count**, **Recorded Memory**, **Dead Count**, or **Dead Memory**.

3. Select from the list a class that has data in the Recorded or Dead columns and click **Call Traces**  .

   The Call Traces view opens.

   Note The **No Recorded Instances** dialog box appears when the selected class contains no recorded instances (that is, when either the JVM did not provide traces for the instances, or trace collection was not enabled when the instances were allocated).



Understanding the Call Traces View

The Call Traces view provides you with several ways of viewing the same method data. It is a good starting point for investigating which methods allocated what instances.

The upper pane contains two tabs:

- **Merged Traces Tree**—is a traditional program call tree that shows methods that are involved in the allocation (that is, it lists the call tree of the program for the call paths that allocated instances that are still live).
- **Traces Graph**—is a traditional call graph. It displays a call graph of the program for call paths that allocated instances that are still live.

When the graph expands beyond the size of the pane, you can use the Panner to focus on various areas of the graph or use the scrollbars to move around the graph. You can also use the Scale slider below the graph to enlarge or shrink the graph.

When you click a node in the graph, the corresponding line in the table (lower pane) is highlighted, and vice versa. The Instance Detail toolbar button then becomes available for further investigation of the instance.

The default status color scheme for the Traces Graph is *Cumulative Count*. You can select an alternate scheme (*Method Count)* from the **Color to Show** list above the graph. You can hover over the color scale at the bottom of the graph to see what they represent.

The lower pane displays a list of methods that have allocated live instances during a session. The table collects data for the methods in the tree or graph into a single entry. Data is displayed in the following columns:

Data	Description
Name	For each row, displays the names of the locations that created the objects. Depending upon how the data is viewed, the name represents method, class, or package data.
Cumulative Count	For each row, displays the number of live instances allocated by the location directly or indirectly by calling other methods.
Count	For each row, displays the number of live instances allocated directly by the location.

The tree, graph, or table may group multiple nodes by category and display just the category name to help you focus on the methods that matter to you. You can drill down into these grouped items by choosing them in the table or tree and clicking **Replace Category with More Detail**  on the toolbar.

To investigate which objects were allocated by a selected package, class, or method, click **Instance Detail View**  . For more information about using the Instance Detail view, see "Exploring the Instance Detail View" on page 166.

Customizing the Call Traces View

You can change the memory units displayed in the method list (**Tools** > **Edit Options** > **Data Display** > **Memory**). You can also customize your Call Traces view by changing the font family, size, and style, as well as the color scheme.

- "Changing the Scale Gradation" on page 177
- "Changing the Color Scheme" on page 177
- "Customizing Fonts" on page 178

Changing the Scale Gradation

By default, the scale gradation is geometric, which means that (except for the first two ranges) the range of the segments doubles from minimum to maximum. For example, if the first segment spans 15 units, the second spans 15, the third 30, the fourth 60, and so on, doubling in size until the maximum value is reached. Use a geometric gradation to see the relative relationships among expensive nodes in the graph, in particular when a graph has a few nodes with values that far exceed the other nodes.

There will be times, however, when you want a quantitative view of the graph. In this case, you can change the scale gradation to linear; the color scale is divided into segments of equal size, and the graph highlights the few, most expensive nodes.

To select the gradation used for the scale:

- To highlight quantitative differences among the nodes, right-click the color scale and select **Linear Mapping**.
- To highlight relative relationships among the nodes, right-click the color scale and select **Geometric Mapping** (default).

Changing the Color Scheme

You can select from a few predefined color schemes. In all schemes, the bright color represents the most expensive methods while the dull color is for the least expensive method. Select a color scheme that suits your preference. If you want to print a graph in black and white, the best scheme to use is **Blue to Light Gray**.

To change the color scheme:

- Right-click the color scale and select a color scheme.

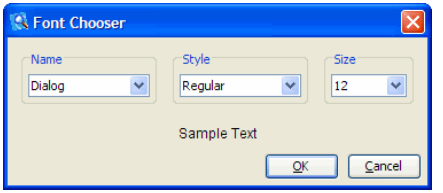
Customizing Fonts

Use the Font Chooser dialog box to change the font family, size, and style of text in the graph.

To change fonts:

1. On the toolbar, click  to open the toolbar menu and select **Set Font**.

   The *Font Chooser* dialog box opens.



2. Select a new font from the **Name** list.
3. Select Regular, Bold, or Italic from the **Style** list.
4. Select a font size from the **Size** list.

   Your choices are displayed in the sample text below the list boxes.
5. Click **OK**.

Interacting with the Call Traces View

The following table summarizes the tasks that you can perform:

Task	Action	Response
Remove a selected node and its children from the dataset	Select a method and click **Prune**  .	The view is updated to reflect the removal of the selected node and its children from the dataset.

Task	Action	Response
Limit the dataset to a selected node and its children	Select a method and click **Isolate**  .	The view is updated to display only the selected node and its children.
Show the stack frames contained in the selected node	Select a method and click **Replace Category with More Detail**  .	The view is updated to expand the data in this view.
Investigate instances allocated by a method	Select a method and click **Instance Detail**  .	The Instance Detail view opens to display the instances allocated by the method.
Expand the tree to show multiple paths	Select a node, right-click, and select **Expand To Next Branch Point**.	The node expands to show more than one branch.
View the source code for an allocating method	Select a method, right-click and select **Show Allocated At Source**. **Note** If this menu item is disabled, JProbe cannot associate a source file with the class file.	The Source view opens with the source code displayed.You may be prompted to locate the source code file.
Show/hide table columns	Right-click the table and select **Show/Hide Columns**. In the *Show/Hide Columns* dialog box, select the columns to show and click **OK**.	The selected columns are displayed. The rest are hidden.

Task	Action	Response
Resize column widths	To set the column width, drag the edge of the column heading left or right. To see all data in a column, right-click in the column and select **Size <column name> to Fit**. To fit all columns in the viewing area, right-click and select **Size All Columns to Fit**.	The columns are resized.
Rearrange columns	Drag a column by its heading and drop it in a new location. **Note** Cannot be used on tree tables.	The column appears in its new location.
Sort the table by a metric	Click the column heading for that metric. To reverse the sort order, click the heading again.	The table is sorted.
Export the table to CSV	Right-click in the table and choose **Export to CSV**. Type a file name and click **Save**.	The file contains the table data separated by a comma. It may also contain headers. For more information, see "Changing How JProbe Exports to a CSV File" on page 455.
Create a report for this view	Click **Export Data to HTML**  .	The *Create Call Traces Report* dialog box appears. For details, see "Call Traces Report" on page 207.

Exploring the Merged Allocation Points View

The Merged Allocation Points view displays a merged stack trace for a set of instances. The merged stack traces lead you to the instance that allocated many objects over the runtime of the program. By default, the stack traces are collapsed and only the allocating methods appear. In order to view merged stack traces you need to run the session with the Collect Stack Traces option chosen.

Note This view is not available if you have set the Investigate By option to Heap Count or Heap Memory in the Instances view. For more information, see "Understanding the Instances View" on page 139..

This section contains the following information:

- "Opening the Merged Allocation Points View" on page 181
- "Understanding the Merged Allocation Points View" on page 181
- "Interacting with the Merged Allocation Points View" on page 182

Opening the Merged Allocation Points View

The Merged Allocation Points view is one of the views where you can focus on a class selected in the Instances view.

To open the Merged Allocation Points view:

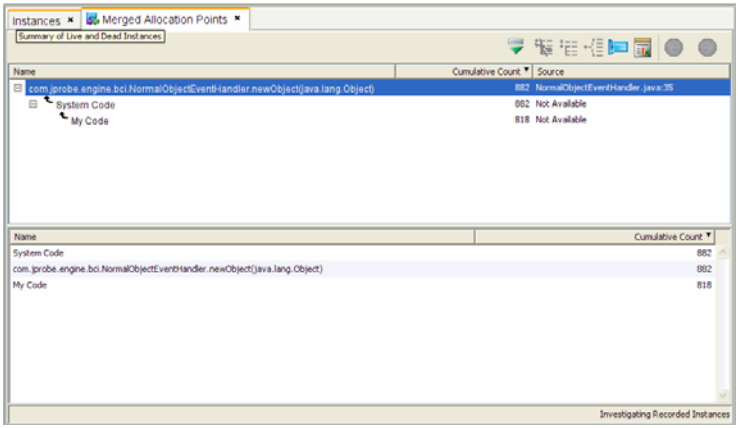
1. In the Instances view, select **Recorded Count** or **Recorded Memory** from the *Investigate By* list.
2. Select a class that has instances.
3. Click **Merged Allocation Points**  .

   The Merged Allocation Points view opens, displaying with the merged stack traces for the selected class. If the message "Empty Dataset: no matches" appears, either the JVM did not provide traces for the instances, or trace collection was not enabled when the instances were allocated.

Understanding the Merged Allocation Points View

The Merged Allocation Points view is a good starting point for investigating where instances originate. Each method that allocates instances of the selected class becomes

the root of a tree. The tree represents the merged stack trace for the method. A merged stack trace includes all methods that called that particular allocating method. Beside each allocating method you can see the number of instances allocated by the method in the Count column.



Note You may see the same method twice in the list. Duplicate allocating methods occur when two lines of code in the method each allocate instances.

When you expand a stack trace, you can see the series of method calls that led to the allocating method being invoked. The arrows beside the method names show the direction of the calls toward the allocating method. Where more than one series leads to the allocating method, each series is shown as a branch in the tree. The value in the Count column for the allocating method is the total of all branches.

If the Merged Allocation Points view shows “Empty dataset: No matches” either the class has no instances of traces, or allocation point data was not collected at the time that the instances of the class were allocated. Stack traces are only collected on instances that were allocated after you started recording data.

Interacting with the Merged Allocation Points View

You can expand and collapse trace entries, as well as review the instances allocated by the method by opening the Instance Detail view. Where you have access to the source code, you can also drill down to the code for either the allocating method or the selected class. If the Source column is blank and/or the Allocated At Source option is disabled,

you do not have access to the source code for this method. If the Class Source option is disabled, you do not have access to the source code for the class.

The following table summarizes the tasks that you can perform:

Task	Action	Response
Expand the tree	Select a node, right-click, and select **Expand to Next Branch Point**.	The node expands to the next node.
Examine the instances allocated by a method	Click **Instance Detail View**  .	The Instance Detail view opens with all the instances allocated by this method in the Instance List.
View the source code for an allocating method	Select a method, right-click, and select **Show Allocated At Source**. **Note** If this menu item is disabled, JProbe cannot associate a source file with the class file.	The Source view opens with the source code for the allocating method. You may be prompted to locate the source code file.
Drill down into a category node	Select the node and click **Replace Category with More Detail**  .	The node expands to one or more nodes.
Remove a node from the tree	Select the node and click **Prune**  .	The path, starting at the selected node, is removed from the tree.
Focus on a single path	Select the node and click **Isolate**  .	Only the paths through the selected node are displayed.
Show/hide table columns	Right-click the table and select **Show/Hide Columns**. In the *Show/Hide Columns* dialog box, select the columns to show and click **OK**.	The selected columns are displayed. The rest are hidden.

Task	Action	Response
Resize column widths	To set the column width, drag the edge of the column heading left or right. To see all data in a column, right-click in the column and select **Size <column name> to Fit**. To fit all columns in the viewing area, right-click and select **Size All Columns to Fit**.	The columns are resized.
Rearrange columns	Drag a column by its heading and drop it in a new location.	The column appears in its new location.
Sort the table by a metric	Click the column heading for that metric. To reverse the sort order, click the heading again.	The table is sorted.
Export the table to CSV	Right-click in the table and select **Export to CSV**. Type a file name and click **Save**.	The file contains the table data separated by a comma. It may also contain headers. For more information, see "Changing How JProbe Exports to a CSV File" on page 455.
Create a report for this view	Click **Export Data to HTML**  .	The *Create Merged Allocation Points Report* dialog box appears. For details, see "Merged Allocation Points Report" on page 208.

Exploring the Heap Graph View

When you want to visualize the references among instances in the heap as a directed graph, you can use the Heap Graph view. Each node in the graph is an instance, and the edges represent the references from one instance to another. The table below the graph is similar to the one in the Instance Detail view; it lists the instances in the graph, the number of referrers to and references from it, and its size.

This section contains the following information:

- "Opening the Heap Graph" on page 185
- "Understanding the Heap Graph" on page 186
- "Interacting with the Heap Graph" on page 187

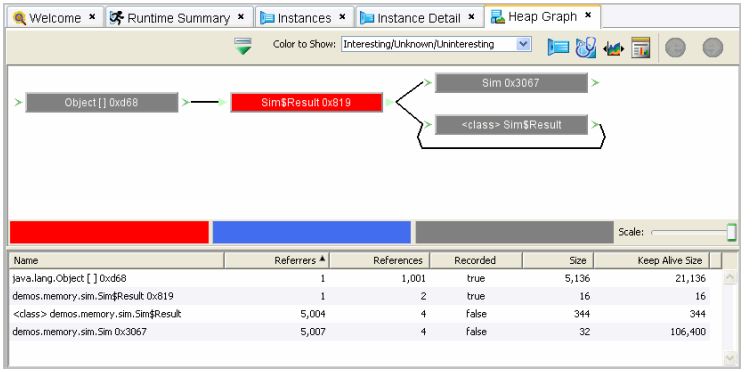
Opening the Heap Graph

The Heap Graph provides a visual map of the referrers and references of a selected instance. Heap Graph view is available from the Instance Detail view.

To open the Heap Graph:

1. Select an instance in the Instance Detail view.
2. Click **Heap Graph**  .

   The Heap Graph opens. The upper pane displays a graph of the selected instance and its immediate referrers and references, and the lower pane displays them in a table.



Understanding the Heap Graph

The Heap Graph initially shows only instances or classes that are directly connected to the selected instance, but you can expand the graph in the following ways:

- Double-click the arrowheads ➤ to the left or right of a node.
- Select a node, right-click, and select **Show Referrers** or **Show References**.

When the graph expands beyond the size of the pane, you can use the Panner to focus on various areas of the graph or use the scrollbars to move around the graph. You can also use the Scale slider below the graph to enlarge or shrink the graph.

When you click a node in the graph, the corresponding line in the table below is highlighted, and vice versa. The Instance Detail and Leak Doctor toolbar buttons then become available for further investigation of the instance.

The default status color scheme for the Heap Graph is *Interesting/Unknown/Uninteresting*. You can select an alternate scheme (*Created When Recording/Not Recording* or *Keep Alive Size*) from the **Color to Show** list above the graph. You can hover over the color scale at the bottom of the graph to see what they represent. You can change the status of a node by selecting it, right-clicking, and selecting one of the options under **Mark As** or **Mark Reachable As**. If you choose the latter, the changed status will be applied to all the nodes below this one in the graph. This can help you to keep track of instances that you have already investigated and have decided are definitely not involved in your leak.

Customizing the Heap Graph

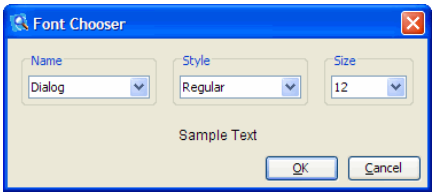
You can change the memory units displayed in the method list (**Tools** > **Edit Options** > **Data Display** > **Memory**). You can also customize your Heap Graph view by changing the font family, size, and style.

Use the Font Chooser dialog to change the font family, size, and style of text in the graph.

To change fonts:

1. In the toolbar, click  to open the toolbar menu and select **Set Font**.

   The *Font Chooser* dialog box opens.



2. Select a new font from the **Name** list.
3. Select Regular, Bold, or Italic from the **Style** list.
4. Select a font size from the **Size** list.

   Your choices are displayed in the sample text below the list boxes.
5. Click **OK**.

Interacting with the Heap Graph

The following table summarizes the tasks that you can perform:

Task	Action	Response
Change the color scheme of the graph	Select an option from the *Color to Show* list.	The graph changes to the other scheme.
Enlarge or reduce the graph	Move the button on the **Scale** slider.	The graph becomes larger or smaller.

Task	Action	Response
Move to a section of the graph that is not in the viewing area	Use the scroll bars to move to the hidden section. or Click **Graph Panner**  and move the rectangle to the desired area.	The graph viewing area shows the selected area.
View details of the selected instance	Click **Instance Detail**  .	The Instance Detail view opens.
Investigate the selected node	Click **Leak Doctor**  .	The Leak Doctor view opens.
View the immediate dominator of a selected instance	Select a node in the graph, right-click, and select the **Show Immediate Dominator** option from the list.	The graph expand and displays the immediate dominator of the selected method.
Change the status of the selected node	Select the node, right-click, and select one of the **Mark As** or **Mark Reachable As** options.	The color of the node or the node and all nodes reachable from it changes.
Show/hide table columns	Right-click the table and select **Show/Hide Columns**. In the *Show/Hide Columns* dialog box, select the columns to show and click **OK**.	The selected columns are displayed. The rest are hidden.

Task	Action	Response
Resize column widths	To set the column width, drag the edge of the column heading left or right. To see all data in a column, right-click in the column and select **Size <column name> to Fit**. To fit all columns in the viewing area, right-click and select **Size All Columns to Fit**.	The columns are resized.
Rearrange columns	Drag a column by its heading and drop it in a new location.	The column appears in its new location.
Sort the table by a metric	Click the column heading for that metric. To reverse the sort order, click the heading again.	The table is sorted.
Export the table to CSV	Right-click in the table and select **Export to CSV**. Type a file name and click **Save**.	The file contains the table data separated by a comma. It may also contain headers. For more information, see "Changing How JProbe Exports to a CSV File" on page 455.
Create a report for this view	Click **Export Data to HTML**  .	The *Create Heap Graph Report* dialog box appears. For details, see "Heap Graph Report" on page 208.

Exploring the Leak Doctor View

After you have identified a potential loiterer, the Leak Doctor displays the references to it as a tree. You can explore the paths and model what would happen if you freed a reference along the reference path. If freeing a reference results in the loitering object being released, you know where you have to fix your code.

This section contains the following information:

- "Opening the Leak Doctor" on page 190
- "Understanding the Leak Doctor" on page 192
- "Interacting with the Leak Doctor View" on page 195

Opening the Leak Doctor

You use the Leak Doctor view after you have identified your loitering object.

To open the Leak Doctor:

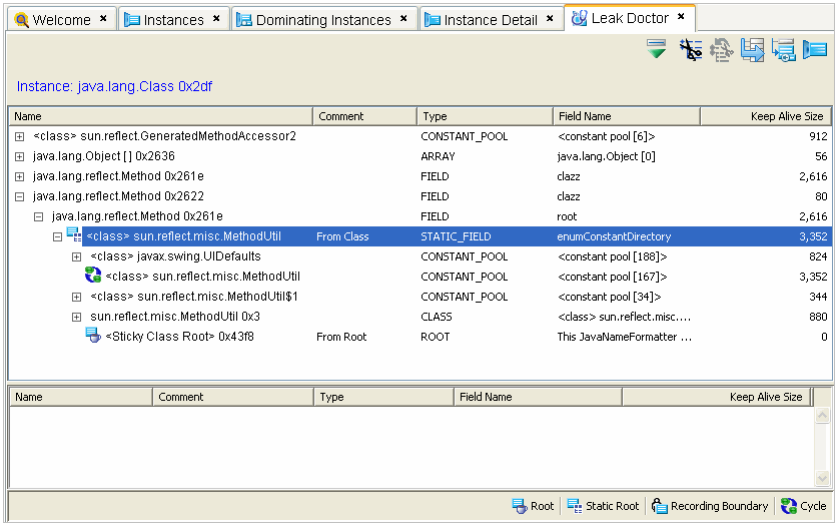
1. In the Instance Detail or Heap Graph view, click the loitering instance.

   or

   Select a group of instances in the Instance Query view.

2. Click **Leak Doctor**  .

   The Leak Doctor view opens, displaying the references to the loitering object.



The upper pane of the Leak Doctor lists references to the loitering object by name, (optional) comment, type of reference, and the field of the referring instance that refers to the object. The toolbar has buttons for removing and restoring edges. The lower pane displays the removed edges.

The name of each field Leak Doctor tree is prefixed by an icon which provides information about that field, as described in the following table.

Icon	Name	Type
	Root	Root instance.
	Static Root	Static root instance.
	Recording Boundary	Instance created at the recording boundary.
	Cycle	Instance that refers to itself (circular reference).

Note The status bar also displays the most common icons and short descriptions of what they represent.

Understanding the Leak Doctor

The purpose of the Leak Doctor is to model what would happen if you removed a reference. You can expand/collapse nodes in the tree, and you can remove edges. After removing references, you can restore them if you like.

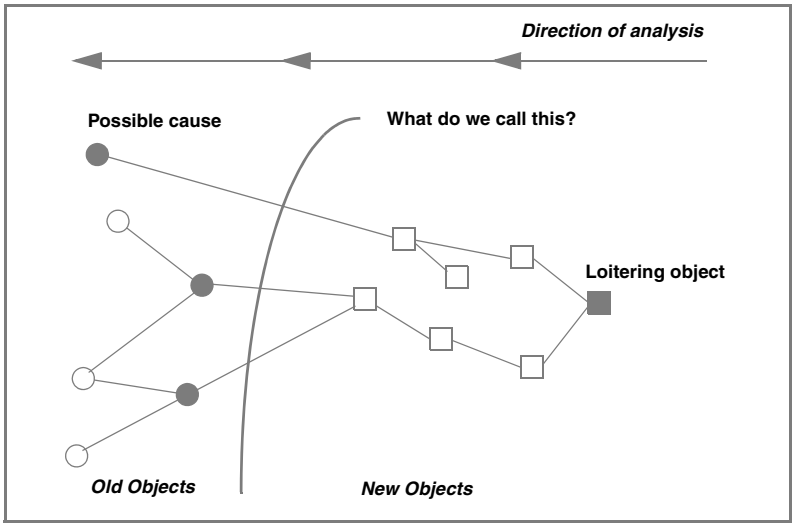
The Leak Doctor starts at the loiterer and traces the paths back through instances that reference the loitering object, using a “breadth first” algorithm. In other words, it looks at all the children of a referrer for objects of interest before moving on to its grandchildren. The search stops when a referrer is discovered that has a comment. The comments are explained in the following table.

Comment	Indicates
From Class	A static field, which can remain in the heap for a long time, possibly housing a problematic collection.

Comment	Indicates
From Root	A root that is responsible for pinning an instance (see a description of root types below the table)
Recording Boundary	A reference from an object that has been around for a long time to an instance that has been around for a short time, which could be problematic
User Code	A transition from a user class to another component which might be worth investigation

A number of instances in the heap will be pinned by roots; the different types are described below.

- JNI Global Roots happen when native code creates a global reference to an instance.
- JNI Local Roots happen when native code is called with instances in the heap.
- Stack Local Roots happen when a method is invoked; they pin the local variables of the method.
- Sticky/System Class Roots happen for a number of classes in the system that the VM does not want to go away.
- Monitor Roots happen when something acquires a lock.
- Thread Roots exist for each thread running in the system.
- Other (not in any of the above categories)



When you choose a reference in the upper pane and click the Remove Edge toolbar button, the reference is removed from the heap and added to the table to the lower pane. If removing this reference frees the loitering object, the following message appears:


Instance Eligible for Garbage Collection
Congratulations, the instance can be garbage collected. Your next step is to free the removed edges in your application code.
OK

You now know that you can remove this reference from your code.

A different message may appear that indicates that no paths to a root were found for this instance. This can be due to a number of reasons, but does not guarantee that this instance is not being held. You may want to investigate further on your own to see if this instance is contributing to the memory problem.

If you have removed multiple references from the graph, it may be good to restore some of them to find the minimal number that are required to free the loiterer. You can restore a reference by selecting it in the lower pane and clicking the Restore Edge toolbar button.

Interacting with the Leak Doctor View

The following table summarizes the tasks that you can perform:

Task	Action	Response
Find the next path to explore	Select the loitering instance and click **Find Next Interesting Path**  .	The tree expands to the next commented reference. Continue to do this until you locate the cause of the loitering instance.
Remove a reference to the loitering object	Click **Remove Edge**  .	The referrer is moved to the lower pane.
Restore a reference to the loitering object	Click **Restore Edge**  .	The referrer is restored to the upper pane.
Show the Instance Details view for the selected edge	Click **Instance Detail**  .	The Instance Detail view is displayed.
Show the path to the nearest interesting dominator	Click **Find Next Interesting Dominator**  .	The tree expands and displays the nearest interesting dominator.
Show/hide table columns	Right-click the table and select **Show/Hide Columns**. In the *Show/Hide Columns* dialog box, select the columns to show and click **OK**.	The selected columns are displayed. The rest are hidden.

Task	Action	Response
Resize column widths	To set the column width, drag the edge of the column heading left or right. To see all data in a column, right-click in the column and select **Size <column name> to Fit**. To fit all columns in the viewing area, right-click and select **Size All Columns to Fit**.	The columns are resized.
Sort the table by a metric	Click the column heading for that metric. To reverse the sort order, click the heading again.	The table is sorted.
Export the table to CSV	Right-click in the table and select **Export to CSV**. Type a file name and click **Save**.	The file contains the table data separated by a comma. It may also contain headers. For more information, see "Changing How JProbe Exports to a CSV File" on page 455.

Exploring the Memory Source View

The Source view displays source code. The source code contains line numbers taken from the line data table that was created when you compiled your code.

This section contains the following information:

- Opening the Source View
- Interacting with the Source View

Opening the Source View

The Source view displays the source code for the method that allocated the instances of a class. If you have the source code for a class, you can open the Source view from the following views: Instance Detail, Call Traces, and Merged Allocation Points.

To open the Source view and display class source code:

1. Click a class or instance of a class from a view that has access to the Source view.
2. Right-click the selected class/instance and select **Allocated At Source**.

   The Source view opens with code for the selected class displayed.

Interacting with the Source View

You can only view your source code in this view; you must edit your source code outside of JProbe.

Exploring the Memory Difference View

The Memory Difference view summarizes the differences between two related memory snapshots. The differences are expressed in terms of the class-level metrics available in both the Runtime Summary and the Instances view: Recorded, Heap, and Dead counts and memory.

Note To get meaningful results, you must compare similar memory snapshots. This means that the snapshots need to be taken at the same relative point in each session.

This section contains the following information:

- Opening the Memory Difference View
- Understanding the Memory Difference View
- Using Filters in the Memory Difference View
- Interacting with the Memory Difference View

Opening the Memory Difference View

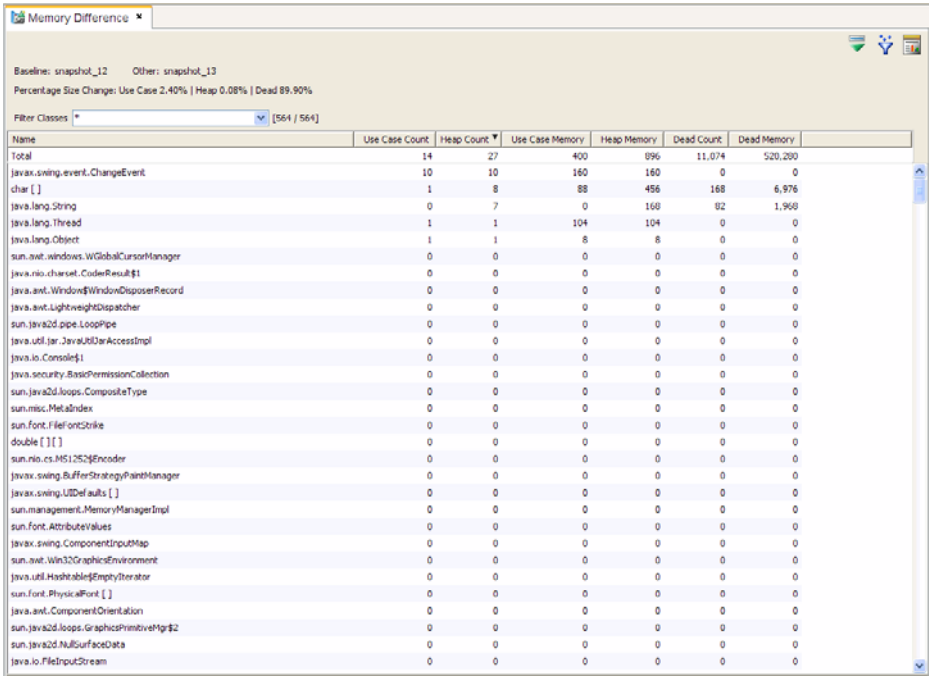
You can open the Memory Difference view if you have loaded at least two memory snapshots.

To open the Memory Difference view:

1. Load the snapshots you want to compare.
2. Right-click the selected snapshot and select **Snapshot Differencing**.

   The *Memory Difference* dialog box opens. The Snapshot to Compare field displays the name of the snapshot you selected.
3. From the *Baseline Snapshot* list, select the snapshot you want to use as the basis for your comparison.
4. In the *Snapshot to Compare* list, either leave the currently selected snapshot or select a different snapshot to compare against the baseline.
5. Click **OK**.

   The Memory Difference view opens.


Memory Difference
Baseline: snapshot_12 Other: snapshot_13
Percentage Size Change: Use Case 2.40% | Heap 0.08% | Dead 89.90%
Filter Classes *
[564 / 564]

Name	Use Case Count	Heap Count	Use Case Memory	Heap Memory	Dead Count	Dead Memory
Total	14	27	400	896	11,074	520,280
javax.swing.event.ChangeEvent	10	10	160	160	0	0
char [ ]	1	8	88	456	168	6,976
java.lang.String	0	7	0	160	82	1,968
java.lang.Thread	1	1	104	104	0	0
java.lang.Object	1	1	8	8	0	0
sun.awt.windows.WGlobalCursorManager	0	0	0	0	0	0
java.nio.charset.CoderResult$1	0	0	0	0	0	0
java.awt.Window$WindowDisposerRecord	0	0	0	0	0	0
java.awt.LightweightDispatcher	0	0	0	0	0	0
sun.java2d.pipe.LoopPipe	0	0	0	0	0	0
java.util.jar.JavaUtilJarAccessImpl	0	0	0	0	0	0
java.io.Console$1	0	0	0	0	0	0
java.security.BasicPermissionCollection	0	0	0	0	0	0
sun.java2d.loops.CompositeType	0	0	0	0	0	0
sun.misc.MetaIndex	0	0	0	0	0	0
sun.font.FileFontStrike	0	0	0	0	0	0
double [ ] [ ]	0	0	0	0	0	0
sun.nio.cs.MS1252$Encoder	0	0	0	0	0	0
javax.swing.BufferStrategyPaintManager	0	0	0	0	0	0
javax.swing.UIDefaults [ ]	0	0	0	0	0	0
sun.management.MemoryManagerImpl	0	0	0	0	0	0
sun.font.AttributeValues	0	0	0	0	0	0
javax.swing.ComponentInputMap	0	0	0	0	0	0
sun.awt.Win32GraphicsEnvironment	0	0	0	0	0	0
java.util.Hashtable$EmptyIterator	0	0	0	0	0	0
sun.font.PhysicalFont [ ]	0	0	0	0	0	0
java.awt.ComponentOrientation	0	0	0	0	0	0
sun.java2d.loops.GraphicsPrimitiveMgr$2	0	0	0	0	0	0
sun.java2d.NullSurfaceData	0	0	0	0	0	0
java.io.FileInputStream	0	0	0	0	0	0

Understanding the Memory Difference View

The Memory Difference view is similar to the Instances view, except that it displays differences between one snapshot and another.

Values

In the table, a negative value means that you improved memory use—fewer instances were created or less memory was used in the compared snapshot. A positive value means memory use worsened since the baseline snapshot—more instances of the class were created in the compared snapshot. A zero value indicates no change. Across the top of the table is a totals line, which shows the overall change the underlying data set.

Note If you have used the Filter Classes option, some of the classes may not be visible.

Percentages

The percentages above the table are calculated using the following formula:
`Percentage = (Compared - Baseline) / Baseline * 100.`

If the resulting value is >1000% or <-1000%, the change is reported as “huge.”

Using Filters in the Memory Difference View

If you want to specify filters, you can use the Set Filters dialog. You can re-use existing filters or de-select them so that they do not filter data. You can also use the Filter Classes option above the table, which is described under the Instance view (see “Filtering Data in the Instances View” on page 141).

To see how data is affected by Set Display Filters, consider the following example of how the Memory Usage Change By percentage is calculated, first with filters applied, then without. The totals in the snapshots are different. For example, the baseline snapshot with filters has a total of 50 instances, while the total without filters is 100 instances.

Memory	Filtered Data	All Data
Total in Baseline Snapshot	50	100
Total in Compared Snapshot	40	90
Usage Change Calculation	((40 - 50) / 50) * 100	((90 - 100) / 100) * 100
Usage Change Percentage	-20%	-10%

Therefore, when comparing Memory Difference results, be sure to take the filters setting into consideration.

Interacting with the Memory Difference View

You interact with the Memory Difference view in much the same way as you interact with the Instances view.

The following table summarizes the tasks that you can perform:

Task	Action	Response
Filter the table	See "Filtering Data in the Instances View" on page 141.	The classes in the table match the filter criteria that you entered.
Show/hide table columns	Right-click the table and select **Show/Hide Columns**. In the *Show/Hide Columns* dialog box, select the columns to show and click **OK**.	The selected columns are displayed. The rest are hidden.
Resize column widths	To set the column width, drag the edge of the column heading left or right. To see all data in a column, right-click in the column and select **Size <column name> to Fit**. To fit all columns in the viewing area, right-click and select **Size All Columns to Fit**.	The columns are resized.
Rearrange columns	Drag a column by its heading and drop it in a new location.	The column appears in its new location.
Sort the table by a metric	Click the column heading for that metric. To reverse the sort order, click the heading again.	The table is sorted.
Export the table to CSV	Right-click in the table and select **Export to CSV**. Type a file name and click **Save**.	The file contains the table data separated by a comma. It may also contain headers. For more information, see "Changing How JProbe Exports to a CSV File" on page 455.

Task	Action	Response
Create a report for this view	Click **Export Data to HTML**  .	The *Create Memory Difference Report* dialog box appears. For details, see “Memory Snapshot Difference Report” on page 208.

Setting Preferences for Memory Views

You set data display options in the *Edit Options* dialog box.

Option	Description
Memory Unit	The options are: • Bytes • Kilobytes • Megabytes

Note “Object size” columns are always displayed in bytes. The Memory Unit setting does not affect them.

For information about setting other preferences in JProbe, see “Setting Preferences” on page 453.

7

Reporting Memory Analysis Results

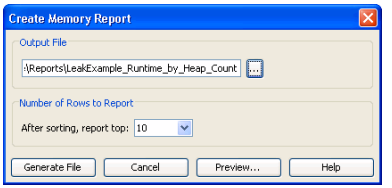
You can generate reports based on most of the Memory views. This chapter presents the Memory reports. To learn the steps to create a report, see “Generating HTML Reports” on page 440.

This chapter contains the following sections:

Setting Report Options 204
Runtime Summary Report 204
Instances Report 204
Dominating Instances Report 205
Pinned Instances Report 205
Allocations Report 206
Allocations Detail Report 206
Instance Detail Report 207
Call Traces Report 207
Merged Allocation Points Report 208
Heap Graph Report 208
Memory Snapshot Difference Report 208

Setting Report Options

When you generate a Memory report, the number of instances in the report is controlled by the Number of Rows to Report option in the Create Memory Report dialog box. The default is the top 10.



Runtime Summary Report

This report contains the contents of the **Data** tab in the Runtime Summary view. Before generating this report, ensure that the data is current by clicking **Refresh Runtime Data** and sort the table by the metric in which you are interested.

Runtime Instances (by Heap Count)

Name	Use Case Count	Use Case Memory (Bytes)	Heap Count	Heap Memory (Bytes)	Dead Count	Dead Memory (Bytes)
char [ ]	462	23,112	5,280	305,728	0	0
java.lang.String	223	5,352	5,038	120,912	0	0
short [ ]	2	32	1,746	228,520	0	0
java.awt.Rectangle	1,431	34,344	1,533	36,792	0	0
byte [ ]	95	11,664	1,485	305,712	0	0
int [ ]	2	544	1,408	144,200	0	0
java.lang.Class	0	0	1,288	548,784	0	0
java.util.HashMap$Entry	49	1,176	968	23,232	0	0
java.lang.Object [ ]	442	11,088	867	33,416	0	0
java.util.Hashtable$Entry	1	24	825	19,800	0	0

Instances Report

This report contains the content of the table in the Instances view. Before generating this report, sort the table by the metric that you are interested in.

Instances (by Heap Count)

Name	Recorded Count	Recorded Memory (Bytes)	Heap Count	Heap Memory (Bytes)	Keep Alive Size (Bytes)	Dead Count	Dead Memory (Bytes)
demos.memory.sim.Sim$Result	5,000	80,000	5,000	80,000	~80,000	0	0
java.lang.String	3	72	2,801	67,224	~186,832	10,060	241,440
char [ ]	0	0	2,666	169,424	~169,424	0	0
java.lang.Class	0	0	1,169	489,944	~774,832	0	0
java.util.HashMap$Entry	0	0	607	14,568	~22,640	81	1,944
java.lang.Object [ ]	0	0	339	40,840	~123,544	0	0
java.util.Hashtable$Entry	1	24	315	7,560	~21,904	0	0
java.lang.Integer	0	0	307	4,912	~4,912	0	0
java.lang.Short	0	0	256	4,096	~4,096	0	0
java.lang.String [ ]	0	0	211	8,344	~17,968	0	0
sun.java2d.loops.Blit	0	0	199	7,960	~7,960	0	0
sun.java2d.loops.ScaledBlit	0	0	163	6,520	~6,520	0	0
java.util.concurrent.locks.ReentrantLock$NonfairSync	0	0	145	3,480	~3,672	0	0
java.util.concurrent.ConcurrentHashMap$Segment	0	0	144	4,608	~12,904	0	0
java.util.concurrent.ConcurrentHashMap$HashEntry [ ]	0	0	144	2,680	~8,264	0	0

Dominating Instances Report

This report contains the content of the table in the Dominating Instances view. Before generating this report, sort the table by the metric that you are interested in.

Pinned Instances Set (by Keep Alive Size)

Name	Recorded Count	Recorded Memory (Bytes)	Heap Count	Heap Memory (Bytes)	Keep Alive Size (Bytes)
java.lang.Integer	365	5,840	365	5,840	~5,840

Dominating Instances (by Keep Alive Size)

Name	Recorded Count	Recorded Memory (Bytes)	Heap Count	Heap Memory (Bytes)	Keep Alive Size (Bytes)
java.lang.Integer [ ]	256	4,096	256	4,096	~4,096
java.util.HashMap$Entry	75	1,200	75	1,200	~1,200
java.util.Hashtable$Entry	17	272	17	272	~272
java.lang.Class	6	96	6	96	~96
java.lang.Object [ ]	5	80	5	80	~80
java.util.Hashtable$Entry [ ]	4	64	4	64	~64
sun.swing.SwingUtilities2$AATextInfo	1	16	1	16	~16
demos.memory.leakexample.LeakExample	1	16	1	16	~16

Pinned Instances Report

This report contains the content of the table in the Pinned Instances view. Before generating this report, sort the table by the metric that you are interested in.

Dominating Instances (by Keep Alive Size)

Name	Recorded Count	Recorded Memory (Bytes)	Heap Count	Heap Memory (Bytes)	Keep Alive Size (Bytes)
java.util.HashMap$Entry	882	21,168	882	21,168	~17,728

Pinned Instances (by Keep Alive Size)

Name	Recorded Count	Recorded Memory (Bytes)	Heap Count	Heap Memory (Bytes)	Keep Alive Size (Bytes)
java.lang.String	47	1,128	721	17,304	~37,480
char [ ]	62	3,488	565	27,584	~27,584
java.lang.Class	--	--	153	60,056	~10,424
java.util.Hashtable$Entry	129	3,096	130	3,120	~3,296
java.util.HashMap$Entry	115	2,760	115	2,760	~3,168
sun.java2d.loops.Blit	42	1,680	42	1,680	~1,680
java.lang.String [ ]	21	808	24	880	~1,536
sun.java2d.loops.ScaledBlit	32	1,280	32	1,280	~1,280
java.util.HashMap$Entry [ ]	6	736	6	736	~1,136
sun.misc.Launcher$AppClassLoader	--	--	1	72	~656

Allocations Report

This report contains the content of the table in the Allocations view. Before generating this report, sort the table by the metric that you are interested in.

Allocations (by Method Count)

Name	Method Count	Method Memory (Bytes)	Cumulative Count	Cumulative Memory (Bytes)
javax.swing	48	936	155	7,040
java.awt	25	512	158	7,120
java.lang	24	904	26	960
demos.memory.leakexample	14	3,528	126	5,880
javax.swing.event	14	448	14	448
java.beans	7	112	14	224
javax.swing.plaf.basic	7	168	132	5,976
java.lang.reflect	7	112	7	112
sun.java2d.d3d	6	304	13	536
sun.java2d	4	160	8	280
java.util	3	88	3	88
sun.swing	2	64	18	720
sun.awt.image	2	264	16	848
java.lang.ref	1	32	1	32
sun.java2d.pipe	1	32	1	32
javax.swing.plaf.metal	1	24	26	912
java.security	1	24	1	24
sun.awt.windows	1	32	1	32
sun.awt	0	0	8	152
java.awt.event	0	0	23	1,064
java.awt.image	0	0	3	288
sun.awt.im	0	0	1	32

Allocations Detail Report

This report contains the content of the table in the Allocations Detail view. Before generating this report, sort the table by the metric that you are interested in.

Allocations Detail (by Count)

Name	Count	Memory (Bytes)
javax.swing.event.EventListenerList	14	224
javax.swing.AbstractButton$Handler	7	112
javax.swing.DefaultButtonModel	7	280
javax.swing.InputMap	7	112
javax.swing.ActionMap	7	112
javax.swing.event.ChangeEvent	6	96

Instance Detail Report

This report contains the contents of the upper table in the Instance Detail view. Before generating this report, sort the table by the metric that you are interested in.

Instance Detail (by # Referrers)

Name	Size	Allocation Time	# Referrers	# References	Allocated At
javax.swing.JButton 0X2513	448	00:05:974	3	25	LeakExample.addButtonToPanel()
javax.swing.JButton 0X2be6	448	00:06:846	3	25	LeakExample.addButtonToPanel()
javax.swing.JButton 0X2db2	448	00:07:102	3	25	LeakExample.addButtonToPanel()
javax.swing.JButton 0X23ce	448	00:05:766	3	25	LeakExample.addButtonToPanel()
javax.swing.JButton 0X2a32	448	00:06:590	3	25	LeakExample.addButtonToPanel()
javax.swing.JButton 0X1f58	448	00:05:383	3	25	LeakExample.addButtonToPanel()
javax.swing.JButton 0X30c2	448	00:07:342	3	25	LeakExample.addButtonToPanel()
javax.swing.JButton 0X2675	448	00:06:166	3	25	LeakExample.addButtonToPanel()
javax.swing.JButton 0X27ec	448	00:06:374	3	25	LeakExample.addButtonToPanel()
javax.swing.JButton 0X22a4	448	00:05:583	3	25	LeakExample.addButtonToPanel()

Call Traces Report

This report lists the locations of all the instances in the Call Traces view, in order of cumulative count. If you are investigating by Dead Count/Memory, these are the locations of garbage collected instances, otherwise they are the locations of live instances. Before generating this report, sort the table by the metric that you are interested in.

Garbage Collected Locations (by Cumulative Memory)

Name	Cumulative Count	Cumulative Memory	Method Count	Method Memory
System Code	207	4,968	0	0
com.jprobe.engine.bci.NormalObjectEventHandler.newObject (java.lang.Object)	207	4,968	4,968	4,968
My Code	0	0	0	0

Merged Allocation Points Report

This report lists the locations of all the instances in the Merged Allocation Points view, in order of cumulative count. If you are investigating by Dead Count/Memory, these are the locations of garbage collected instances, otherwise they are the locations of live instances. Before generating this report, sort the table by the metric that you are interested in.

Locations (by Cumulative Count)

Name	Cumulative Count	Cumulative Memory (Bytes)
demos.memory.sim.Sim.run()	5,000	80,000
java.lang.Thread.run()	5,000	80,000
demos.memory.sim.Sim$Server.query(int, java.lang.String, java.lang.String, java.lang.String)	5,000	80,000
demos.memory.sim.Sim$Client.start()	5,000	80,000

Heap Graph Report

This report contains the contents of the table in the Heap Graph view. Before generating this report, sort the table by the metric that you are interested in.

Heap Graph (by # Referrers)

Name	# Referrers	# References	In Use Case
javax.swing.plaf.ColorUIResource 0X8caf	67	0	false
sun.swing.PrintColorUIResource 0X972b	41	1	false
java.util.Locale 0X7c26	27	3	false
javax.swing.plaf.InsetsUIResource 0Xa432	25	0	false
sun.awt.AppContext 0X8b80	23	5	false
javax.swing.plaf.FontUIResource 0X8b71	21	2	false
java.awt.ComponentOrientation 0X8cc7	20	0	false
javax.swing.plaf.BorderUIResource$CompoundBorderUIResource 0Xa35f	14	2	false
javax.swing.plaf.metal.MetalButtonUI 0Xa6d6	13	2	false
javax.swing.AbstractButton$Handler 0X22aa	7	1	true

Memory Snapshot Difference Report

This report contains the contents of the Memory Difference view. Before generating this report, sort the table by the metric that you are interested in.

Snapshot Difference (by Heap Count)

Name	Use Case Count	Use Case Memory (Bytes)	Heap Count	Heap Memory (Bytes)	Dead Count	Dead Memory (Bytes)
char [ ]	0	0	1	24	0	0
java.lang.String	0	0	1	24	0	0
sun.java2d.windows.GDIBlitLoops	0	0	0	0	0	0
java.util.Hashtable	0	0	0	0	0	0
java.awt.image.VolatileImage [ ]	0	0	0	0	0	0
sun.java2d.loops.DrawGlyphList	0	0	0	0	0	0
java.nio.charset.CoderResult	0	0	0	0	0	0
java.util.concurrent.locks.ReentrantLock$NonfairSync	0	0	0	0	0	0
java.io.ObjectStreamField [ ]	0	0	0	0	0	0
java.io.Console$1$1	0	0	0	0	0	0

8

Troubleshooting Memory Analysis Results

As you work with JProbe, you may need help to get the most out of your analysis sessions. This chapter presents a few tips for solving some Memory analysis problems and for running sessions efficiently. For Memory analysis issues that are not covered here, refer to Quest's technical support.

This chapter contains the following sections:

Memory Troubleshooting Tips 212
Memory Session Runtime Tips 214
Memory Help Messages 215

Memory Troubleshooting Tips

The following tips can help you with problems you may encounter. See the *ReleaseNotes.htm* file for other product-related issues.

I am having difficulties configuring my environment.

Contact Quest Support for assistance. When you do, be prepared to describe your environment. The customer service representative may ask you to run a JProbe session that includes some debugging options to describe your settings more fully.

The JProbe Console cannot open a large memory snapshot.

The first time JProbe opens a memory snapshot in the JProbe Console, it indexes and sorts the data in the snapshot. This can be a time- and memory-consuming operation, especially for large snapshots. If you encounter a JProbe Console error, try using the `jpprepare` utility to preprocess the snapshot. For more informations, see "jpprepare" under "Utilities" in the *JProbe Reference Guide*.

The session hangs when running an application server under load.

If you notice excessive heap growth without garbage collections followed by an application crash, increase the `-Xms` and `-Xmx` parameters or lighten the load so that the JVM has a chance to catch up and do a garbage collection. As the JVM caches data, you can increase the load.

Note For some JVMs you will need to specify the initial heap size and maximum heap size with a specific value, such as `-Xms512m -Xmx512m`.

For example, the compiling process for JSPs takes up a significant amount of memory. If, for example, your session is failing at a load of only five pages per second, this is probably a memory allocation problem. Running through the JSPs at a slow pace allows the application server to cache them, and afterwards you can increase the load on the system without difficulty.

In the Runtime Summary view, the memory reported in the Memory Pools Summary chart does not match the total line in the Runtime Data tab.

The value reported in the Memory Pools Summary chart represents an approximation of the size of the heap based on summary data received from the JVM about memory used and memory freed. Because it is an approximation, it may not exactly match the total in the **Data** tab.

So, for the purposes of your investigation, use the chart and values in the Runtime Summary as a starting point, but for serious investigation, take a snapshot and use the Instances view to analyze your data.

The Merged Allocation Points or Call Traces view states “Empty data set: No matches.”

If the Merged Allocation Points view indicates “Empty data set: No matches,” this may be because the JVM did not provide traces for any of the objects of the class. Another reason may be that you did not set the runtime session settings to collect traces (if you are investigating by Recorded Count/Memory) or garbage data (if you are investigating by Dead Count/Memory). Click **Specify Settings**  to open the *Set Recording Level* dialog box, then select the appropriate option to collect traces, garbage collection data, or both.

The stack traces are inaccurate in the Merged Allocation Points view.

For example, consider the following Java program:

```
Method A
line 100 calls Method B
..
Method B
line 200 Object o = new Object();
```

Most JVMs do some optimization to the code and eventually can inline the method B into the method A. After inlining, the line number reported in the stack trace will be `A:100`—that is, the line where A calls B, not the line where `Object o` is created. B is not visible in the stack trace.

You will know when you open the source code that inlining has occurred. You should be able to determine which method was inlined from the code associated with the line number, locate that method, and investigate the problem. If you are not familiar enough with your program to locate the method, you can rerun the session without the JVM’s optimizing compiler. Your program will run slower, but the accuracy of the allocation point information should improve because methods are not inlined. For more information, see “Disabling the Just-in-time (JIT) or other Optimizing Compiler” in the *JProbe User Guide*.

Also, due to the way the JVM reports allocations, the trace may be deeper than you expect. For example:

```
Method C
line 300: Widget w = new Widget()
```

It is possible that JProbe will report the allocation as being in the Widget constructor, `Widget.<init>()` instead of method C, but method C will likely be in the trace, calling the Widget constructor.

Memory Session Runtime Tips

The following tips will help you improve session runtime performance, regardless of the type of analysis session you are running.

Try different supported JVMs.

The session runtime performance depends in part on the speed at which your Java program runs. Modern JVMs, such as Java 6, generally improve program performance. For more information, see the JVM documentation and the *Installing JProbe* guide.

Exit your program before investigating snapshots.

If you open a snapshot in one of the investigation tools while your application continues to run, you may find that the JProbe Console responds more slowly than desired. To improve JProbe Console performance, exit your program before opening a snapshot. Alternatively, you can disconnect from the session, if you want your program to continue running.

Choose not to monitor the session.

Monitoring the session requires an extra JVM be running: the JVM running the JProbe Console. You can run a JProbe session without the JProbe Console. Instead of interacting with the session through the JProbe Console, you use triggers to collect data and take snapshots. After the session ends, you can open the snapshots in the JProbe Console.

Alternatively, you can attach a Console, take a snapshots and when finished detach and close the Console.

Do not collect stack traces or garbage data if you do not need them.

Collecting stack traces can be expensive and may not be needed for an initial investigation. If you do need traces or garbage data, use the Set Recording Level dialog to collect traces for only the classes you need.

Reduce load on your program before taking snapshots.

Taking a snapshot is an expensive operation in the JVM. It will be faster and more likely to succeed if your program is idle.

Memory Help Messages

When you try to execute an action that is not enabled, the JProbe Console usually displays a message explaining why that action is currently disabled and suggests what you need to do to resolve the situation.

This section provides a list of memory help message you may encounter while doing a Memory Analysis using JProbe:

- "No Heap Data" on page 215
- "No Stack Trace Data" on page 216
- "Operation Not Valid for Categories" on page 216

No Heap Data

You can execute this operation only if your snapshot contains heap data.

A snapshot with no heap data happens very rarely. It may occur in one of the following cases:

- You have imported into JProbe a snapshot taken with no heap information.

  No workaround exists for this case.

- You have attempted to take a snapshot with a heap dump, but the JVM failed to provide a valid dump.

  Workaround: Try to reduce the load on your application before taking the snapshot, if possible. Let the application and the JVM become as idle as possible (for example, for a Web application, stop making requests), and manually request a garbage collection (GC). Wait until you see the effect of the GC in the Runtime Heap graph, do another manual GC, then wait again. Take a new snapshot.

  Failures sometimes can be "sticky", that is, once a JVM fails to take a heap dump in a run, subsequent attempts may also fail. Therefore, if you get a failure and the workaround suggested here does not help, shut down the application or application server, and restart it, this time trying the workaround suggestions for every snapshot.

No Stack Trace Data

You can view stack traces only if your snapshot contains stack trace data. The type of data collected during a session is defined by setting the Memory Recording Level appropriately.

If you want to collect stack trace data, you must set the Memory Recording Level to one of the following options, then run a new session to obtain a new snapshot.

- *Record Allocations and Stack Traces For All Instances*.
- *Record Allocations, Stack Traces, and Garbage Data For All Instances*.
- *Record Data as Specified in the Table Below* (define filters for problem classes and set the action to **include**).

For more information about stack traces, see "Setting the Recording Level" on page 110 and "Analysis Options for Memory" on page 124.

If you have set the recording level to collect traces and still see the `No Stack Trace Data` message, it is possible that the JVM is not making traces available to the JProbe Analysis Engine. Disabling the JVM's runtime or just-in-time compiler may help (for example, `-Xint` on some JVMs).

Operation Not Valid for Categories

The tree, graph, or table may group multiple nodes by category and display just the category name to help you focus on the methods that matter to you.

Certain operations cannot be performed on categories. You can drill down into these grouped items by selecting them in the table or tree and clicking **Replace Category with More Detail**  on the toolbar.

For more information about how to manage categories, see "Creating and Managing Categories" on page 432.

9

Strategies for Performance Analysis

This chapter describes strategies for using the JProbe Performance analysis tool to uncover performance issues in your application data.

Tip Use JProbe Memory first to investigate how your program uses memory and to streamline garbage collections.

This chapter contains the following sections:

Strategies for Achieving Program Efficiency 218
Understanding the Memory Pools Chart 221
Understanding Performance Data 222

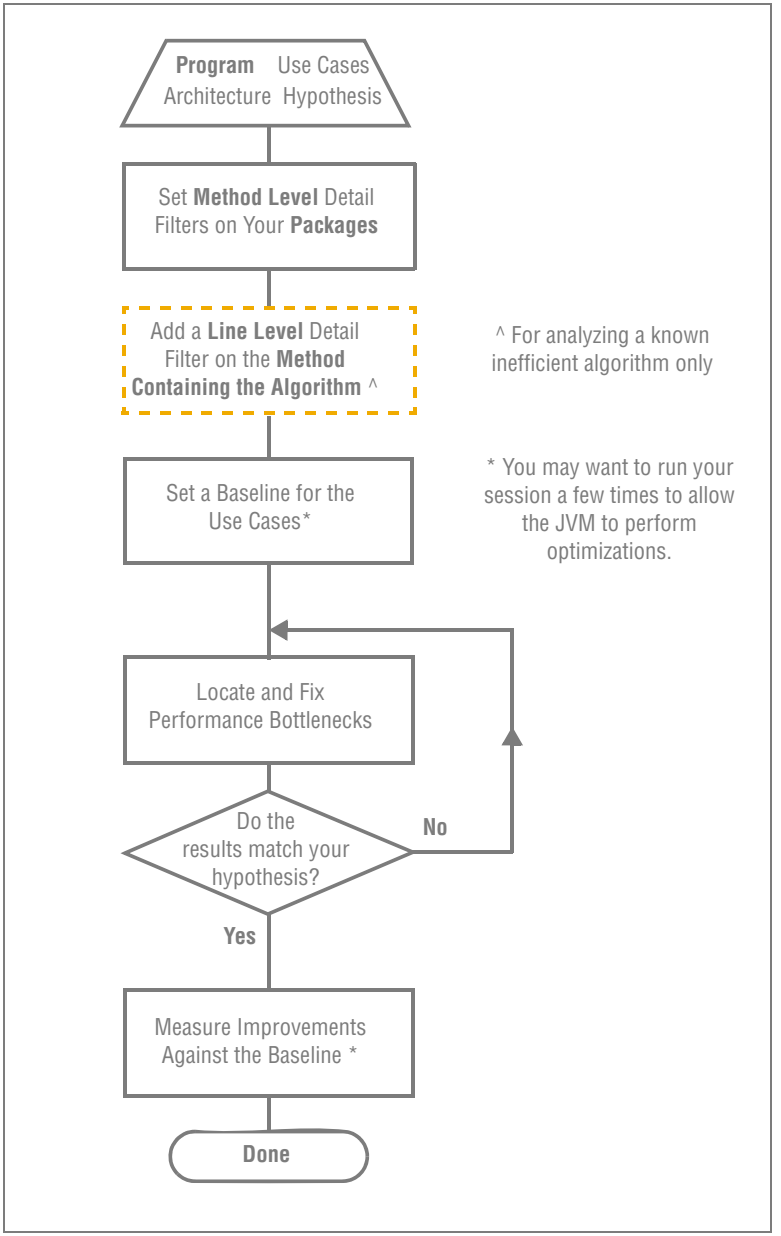
Strategies for Achieving Program Efficiency

There are two common ways to use JProbe Performance:

- Analyzing an Existing Program
- Focusing on a Known Inefficient Algorithm

Analyzing an Existing Program

Senior developers often use JProbe Performance at the end of a development cycle to improve program efficiency. JProbe Performance highlights hotspots in your program, which helps you to identify performance bottlenecks—algorithms that are too slow or that block other processes for too long. To analyze the entire program, you should be an experienced software developer with good overall knowledge of the architecture and expected behavior of the program.


Program Use Cases
Architecture Hypothesis
Set Method Level Detail Filters on Your Packages
Add a Line Level Detail Filter on the Method Containing the Algorithm ^
^ For analyzing a known inefficient algorithm only
Set a Baseline for the Use Cases*
* You may want to run your session a few times to allow the JVM to perform optimizations.
Locate and Fix Performance Bottlenecks
Do the results match your hypothesis?
No
Yes
Measure Improvements Against the Baseline *
Done

While not required, you should take some initial performance snapshots of your program before you implement any changes to it. These snapshots serve as the baseline

against which you assess subsequent code improvements. Setting and measuring against a baseline helps you to quantify improvements to yourself, your team members, quality assurance, product management, marketing, and other stakeholders. Be sure to run your use case several times before selecting a snapshot to use as a baseline. The JVM may perform some optimizations that show improvement only after several runs of the code. For more information, see “Saving Baseline Snapshots” on page 239.

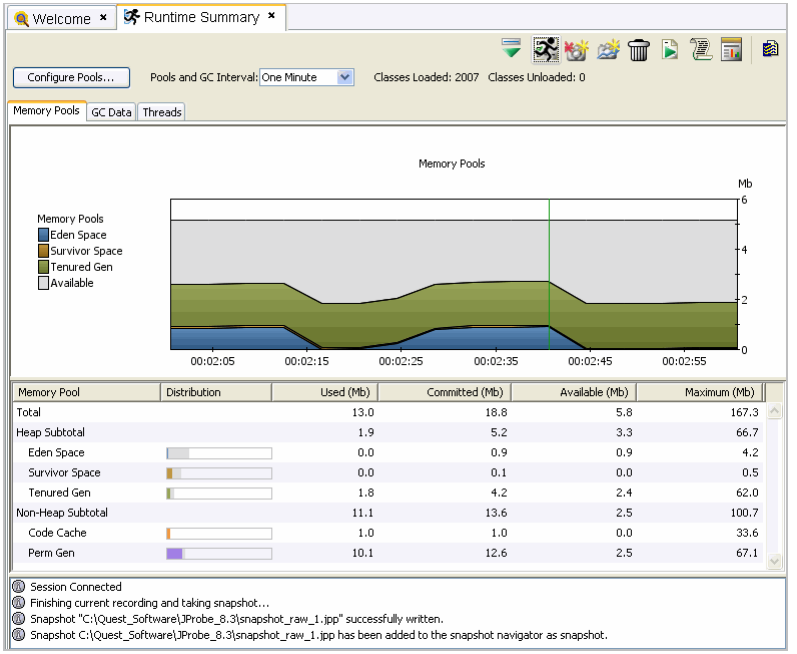
Focusing on a Known Inefficient Algorithm

Any time during a development cycle, a senior software developer may suspect that a heavily-used algorithm is a performance bottleneck, even though it is running as efficiently as possible. You can use JProbe Performance to analyze how the algorithm is called. For the most part, however, you should discourage your team from routinely using JProbe Performance on their algorithms. You only want to spend time optimizing the algorithms that your customers use most frequently; this use assessment is most effective when done by a senior software developer looking at the program as a whole.

While you run your program, Performance keeps track of how long each of your methods takes to execute, either in elapsed (wall clock) time or in CPU time. The Summary view then shows you the hotspots in your program. Hotspots are methods that take a long time to execute and are the first place to look when you are tracking down performance bottlenecks.

Understanding the Memory Pools Chart

In JProbe Performance, the Memory Pools chart acts as a progress bar for the performance of your program, marking how long the program runs in real time. It also provides an overview of how your program uses memory in the Java heap. For example, many tall spikes in the graph represent frequent, large garbage collections. In this case, or if you are curious about memory in use for long periods of time, you can use JProbe Memory to streamline garbage collection activities and improve the way in which your program manages object lifecycles.



When you run a session, the Memory Pools chart displays the available size of the Java heap and the memory in use by your program over time. You can set the time span for the graph. For more information about this view, see “Exploring the Runtime Summary View” on page 274.

Understanding Performance Data

This chapter helps you to gain a better understanding of how data is captured by the Performance analysis tool and displayed in the Methods view. It also introduces the features that you can use to fine tune your analysis and your results.

This section covers the following topics:

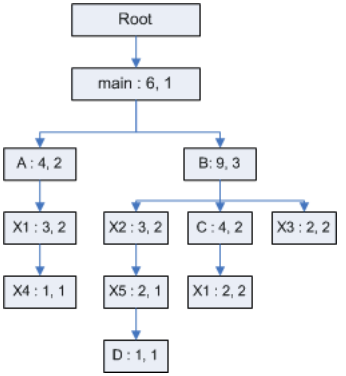
- Collecting Data on Methods
- Building a Calling Context Tree
- How Data is Affected by Using Performance Features
- Recursive Methods

Collecting Data on Methods

The Performance tool captures information on how long methods and lines of data in your application take to process. In order to perform this function, JProbe needs to know when methods are called and when they are returned. JProbe relies on its own instrumentation to generate these events, which it does by interfacing with the JVM and if permissible, modifying classes as they are loaded during the execution of the program.

Building a Calling Context Tree

JProbe then builds a Calling Context Tree (CCT) with the Performance information captured during the execution of the application program. The information in the CCT is not as complete as the Dynamic Call Tree (DCT). In the DCT, each node represents a single call to a method from the method in the parent node, which makes the tree complete, but very large in size. In a CCT, each node has only one parent. This allows the CCT to show a call history of the program in context – or a call context for all metrics.



Simple CCT

How JProbe Builds a Calling Context Tree

JProbe builds a CCT for each thread, and creates a node in the tree to represent the thread. To keep the data as a tree, it also creates the root node, and makes all the thread nodes children of the root node. In the preceding CCT example, there is a single root node that connects to the main node indicating a single thread of execution. Each node has method time and method call numbers associated with it. For example, A:2, 3 represents (A=method name, 2= the method time, and 3= the number of method calls).

Typically, when viewing data in JProbe (after clicking the Include Threads button) you see multiple threads that are being called by the root node. One of these threads calls the main node.

Tip When multiple threads execute simultaneously, sometimes the elapsed time (the program runtime) is less than the method execution time.

The advantage of keeping a separate CCT for each thread becomes clear when using the Prune and Isolate features. For more information about Pruning and Isolating data see “Isolating Subtrees” on page 229 and “Pruning Subtrees” on page 231.

Collected Data

It is important to understand the information that is captured in the CCT. The following are definitions for time captured in the CCT:

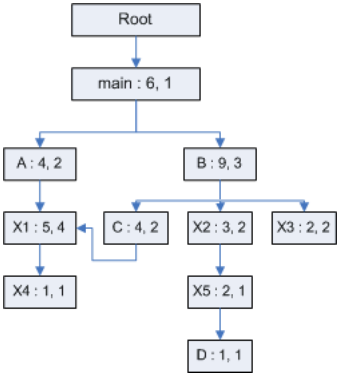
Time Element	Definition
Method Time	The time spent executing code in the method itself; it is sometimes called self time for this reason. In particular, it does not include the time spent executing children it called.
Current Time	The current time (measured according to the settings.)
Start Time	The current time at the start of the method.
Cumulative Time	Cumulative time for a node in the tree is, in general, the sum of method time and child time. Conceptually, it is the total time that the method was active on the stack, or the total time between when the method was entered, and when it exited.
Child Time	The cumulative time of a node's child nodes. The effect of this is that cumulative time accumulates up the tree.

JProbe notes the start time when a method is entered, adding a node to the tree for the method if necessary. When a method is exited, it takes the difference between the current time and the start time and adds this to the node's cumulative time. Only cumulative time is tracked as the program runs; method time is calculated later.

The JProbe Call Graph

The Call Graph is a tool for viewing the CCT to show the calling relationships among methods in your program. The Call Graph can be thought of as a set of superimposed call trees. For example, if two methods call method `A`, then method `A` appears as a single node with two parents. If you remove the call tree for one parent (called pruning), method A remains because it has another parent. The method list located below the graph contains a list of methods with data for all the metrics.

In the Call Graph, times and call counts for the edges are displayed as a tool tip, but the data is not as rich as data found in the CCT.



Call Graph

The JProbe Call Tree

The Call Tree presents the same data as the Call Graph in a traditional, more detailed tree view. While the graph displays only one node per method, the tree view has a node for every distinct call path.

When you use the prune or isolate functions in the Call Graph, the corresponding nodes are removed from the Call Tree. However, if you use those functions in the Call Tree, the Call Graph is not changed, although the underlying data for the corresponding nodes does change. The only exception is if the corresponding node in the graph represents the only call path; in that case, it is removed from the graph.

How Data is Affected by Using Performance Features

When you use features in the Performance tool such as filtering and encapsulation, pruning, and isolating these actions directly affect the data in the CCT. In order to understand these features, refer to the Simple CCT diagram on page 223 as a starting point.

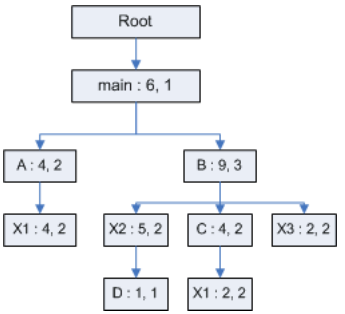
Filtering and Encapsulation

Using the filtering and encapsulation feature allows you to view data that you want to see and minimize data that you do not want to view. For example if you do not want to view data from a third-party source, you can encapsulate particular methods so that you are viewing data that you interested in seeing, therefore creating a boundary between methods that you want to view and those that you do not want to view. For example, you

can add a filter for `*.X*():encapsulated` which results in `X1`, `X2`, `X3`, `X4`, and `X5` to all be considered encapsulated.

Full Encapsulation

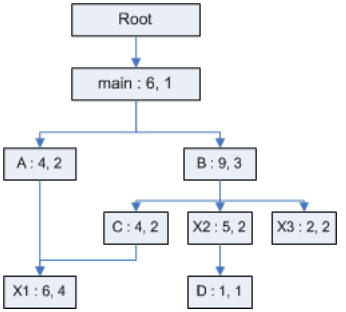
Filtering affects the CCT that the JProbe Analysis Engine collects. The JProbe Analysis Engine connects to the JVM that is executing your application and collects data. Depending upon the JProbe settings that you select (Collect Encapsulated) the results will vary. When the encapsulation type Full is selected, the Performance analysis tool collects the first call to an encapsulated method, but will treat that boundary method as a *black box* with all encapsulated methods it calls. The structure of the tree is preserved and displays information about methods at the boundary, but the tree is larger. With the example and the filter above, the CCT that JProbe collects has the following structure.



Full Encapsulation — CCT

Comparing the Full Encapsulation—CCT to the Simple CCT diagram on page 223, note that `X4` and `X5` are gone in the Full Encapsulation—CCT diagram, because they were originally called by other encapsulated methods. Their time now appears as part of the method time of the encapsulated method that calls them. For example, the node `X4` (that is gone) was called by `X1`. The method time for `X1` is now 4 nanoseconds, which includes 1 nanosecond from the encapsulated `X4`. `X5` is treated in a similar fashion, with its value rolled into `X2`, however the call out to `D` is preserved.

The following diagram, Full Encapsulation—Call Graph, displays how the Call Graph shows this CCT.

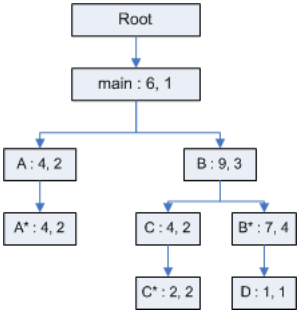


Full Encapsulation — Call Graph

Encapsulated Grouped

When using the Collect Encapsulated Grouped option, all encapsulated children of a non-encapsulated method are combined into one node. For example, the group of encapsulated children of `A` is called by the name `A*` and the group of encapsulated children of `B` is called by the name `B*`, and so on. In the Call Graph, these types of nodes are represented by an icon with an asterisk (*).

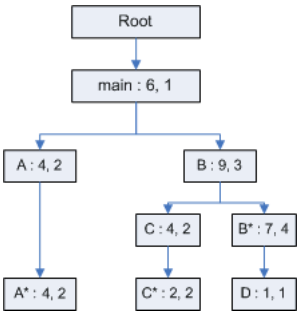
When the Collect Encapsulated Grouped option is selected, the CCT that the Performance tool creates has a similar structure to the structure shown in the Grouped Encapsulation—CCT diagram.



Grouped Encapsulation — CCT

Comparing the Grouped Encapsulation—CCT to the Simple CCT diagram on page 223, the nodes for `X2` and `X3` are combined, and their times and call counts are added together. This type of encapsulation preserves the structure of the CCT and shows the boundary methods but without details.

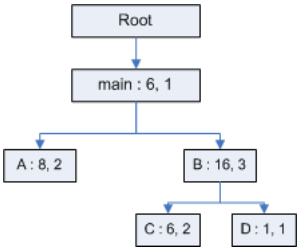
The following diagram shows how the Call Graph represents this CCT.



Grouped Encapsulation — Call Graph

Encapsulated None

Finally, with the Collect Encapsulated option set to None, the encapsulated nodes disappear completely from the CCT, and their times are added into their caller as displayed in the following example.



Encapsulation using None option

This is the fastest type of encapsulation but there is no indication of boundary methods which leads to the situation that you might mistakenly think that a node called another node when really there are several nodes in between the calling node and the child node. For this reason, it is important that you are familiar with your own program. For example, `B` did not call `D`, though it might appear that this is the structure from the tree.

Isolating and Pruning Call Trees

This section describes how isolating and pruning trees affects the data that you see in the Call Graph.

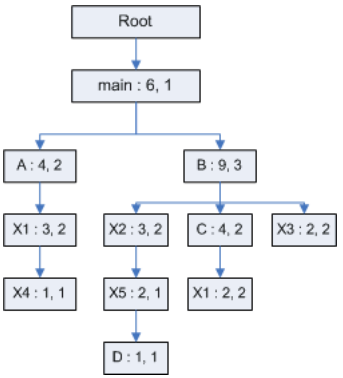
Isolating Subtrees

In the Call Graph, data about all calls to a particular method are combined into a single node. Included in this method is data that you might not be interested in viewing.

For example, consider the following scenario. You are using a Java EE application server and begin to profile code. Your code is called by its `doGet()` method and your methods may call `StringBuffer.append()`. You are interested in how much time was spent by your code in the `append()` method. The application server itself also calls this method, and it is difficult to tell what time is the result of your own code or the use of the application server's.

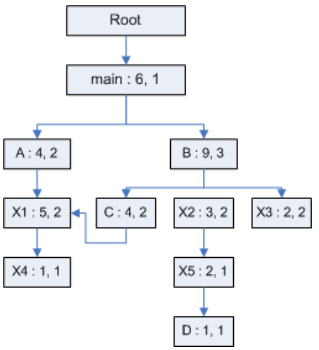
The CCT allows you to remove the application server's use of the append() method by using the Isolate command on the `doGet()` method. The Isolate command makes a new CCT rooted at the selected node.

To illustrate this command, start with the following CCT.



CCT for Isolate

The corresponding Call Graph looks like the following CCT.



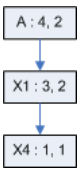
CCT for Isolate via Call Graph

Consider the following example:

```
A= doGet()
B= appServerImpl()
X1= append()
```

In the Call Graph, `X1 (append())` has 5 nanoseconds of method time, but we cannot tell how much of this is due to the code of the application. By isolating `A (doGet())`, a copy of the CCT is created whose top node is `A`. The data for `main, B`, and all of `B's` children including the calls to `X1` down that part of the tree is discarded.

The data appears the same in the CCT or the Call Graph. The isolated tree contains only the times and counts associated with the `doGet()` method. This is just the branch of the CCT rooted at `A`.



CCT rooted at A

The node `X1` has time of 3 nanoseconds, and 2 calls (as a result of the application code). For more information on isolate see “Editing the Data Set” on page 318.

Pruning Subtrees

Pruning removes part of the CCT and therefore changes the shape and alters the values in the graph. This command actually removes data from the Call Graph.

Note This feature differs from the Hide Node option on the graph, which only affects visibility and not the values in the Call Graph.

For example, in your profiling results you see class loading activity. There is little value in viewing the data for the class loader, because you cannot modify its code to improve its performance. By selecting the `loadClass()` method and pruning it, the time spent loading it would be eliminated from your profiling results.

In the following graph image `X4` represents class loading activity.



CCT

By pruning the `X4` node the following results appear.



X4 node pruned

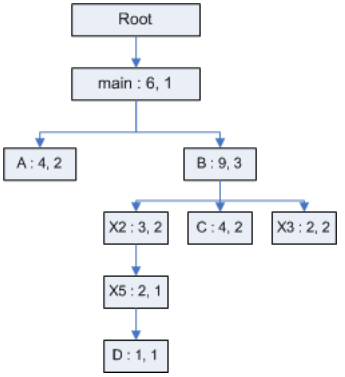
Notice that the cumulative time for the remaining nodes has changed, since the time for `X4` is no longer included.

Isolate and Prune Operations

Prune and Isolate operations affect the current CCT, not the original one. In the proceeding example, method A was isolated, and then additional methods were pruned from its isolated subtree, not from the original CCT.

In this example, an Isolate and then a Prune was performed on the result of the Isolate, not on the original CCT. If for any reason you do not want to save the results, use the **Back** button.

Prune and Isolate Subtree are available from the Call Graph, any of the lists of methods beneath it, or the toolbar. Pruning or Isolating a method that calls multiple nodes affects all the nodes. For example, if you pruned `X1` on the original CCT, both the `X1` node and `X4` node are removed.



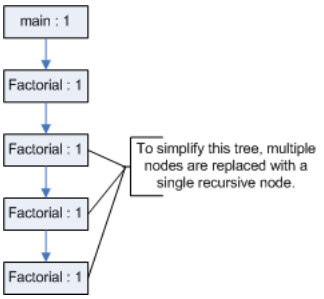
Pruning X1

By using Prune and Isolate commands we can view or exclude data from particular threads. To look at a single thread you can isolate on its thread node. To exclude particular threads (for example, application server threads) you can prune them.

Recursive Methods

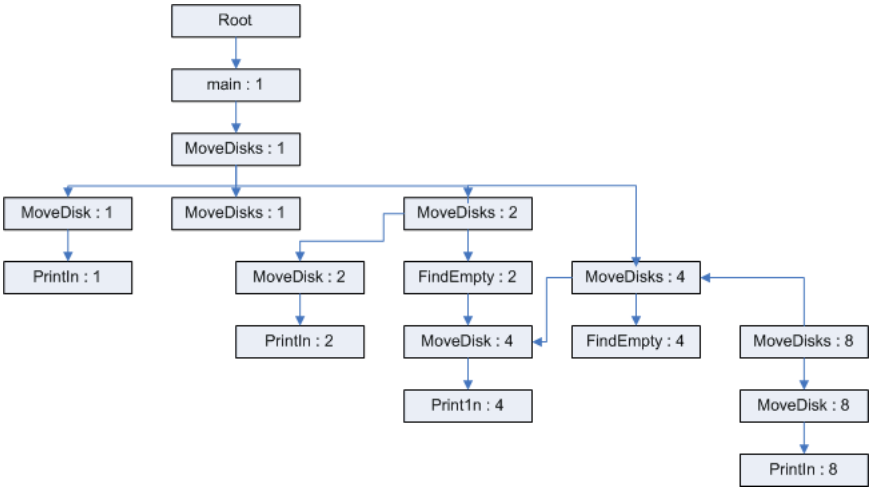
Recursion occurs when a method calls itself, directly or indirectly. In the CCT, this means that a node for that method has a parent or ancestor node for the same method. JProbe saves space by using special recursive nodes.

For example, consider the following CCT for `Factorial`.


main : 1
Factorial : 1
Factorial : 1
Factorial : 1
Factorial : 1
To simplify this tree, multiple nodes are replaced with a single recursive node.

Recursive example showing call count

A more complex example, like the classic Towers of Hanoi puzzle with 4 disks, the call tree showing call count might look like the following example.



Towers of Hanio example

Cumulative Time and Recursion

With our original definition of cumulative time (self time and the cumulative time of children) the cumulative time for `MoveDisks` is encountered twice. If the Recursion example displayed method time instead of call count, the cumulative time for `MoveDisk` would be 30 (15 + 15). Therefore, the cumulative time for `MoveDisk` is 30 + 7 + 14 + 15 = 66.

Essentially the 14 from `MoveDisks(R)` is double-counted because it is included in both the method time, and again in the child time. To resolve this issue, we modify our definition of cumulative time slightly to take recursion into account. The cumulative time for a method is its self time, plus its child time, excluding any recursive calls. Now the cumulative time is 15 + (30 + 7) = 52.

10

Running a Performance Analysis

This chapter describes how to run an analysis session and investigate your application's performance.

This chapter includes the following sections:

Setting Up JProbe for a Performance Analysis Session 236
Setting the Recording Level 236
Running an Analysis Session 237
Investigating Your Results 243

Setting Up JProbe for a Performance Analysis Session

Before you can run a Performance session, you create basic session settings for your application according to the instructions in "Configuring JProbe to Analyze Your Application" on page 23. Later, you can analyze individual methods with line-level precision or measure the time the CPU actually spends on each thread. For more information, see "Narrowing Your Investigation" on page 249.

When you run the JProbe analysis session, your program launches or your application server starts. For programs that require input to run, you now exercise your program. By focusing on areas with reported performance problems, you can be confident that any performance improvements you subsequently implement will be visible to your customers.

Tip The JVM can influence the performance of your program over time as optimizations are made. You may want to run the program for a while, and execute the use case a few times until you begin to get fairly consistent values, and then take the snapshot.

Setting the Recording Level

Before you start your analysis, you need to decide what kind of data you want to see. The *Set Recording Level* dialog box presents the following options.

Select this option	In this situation
Data Recording Off	You are at the initial stage of your investigation, when you do not know what are the performance bottlenecks. This is often enough to at least start your investigation.
Public Methods, No Encapsulation	You collect data only for the public methods matching your filters.
No Encapsulation	You collect data only for the methods matching your filters.
Grouped Encapsulation	You collect data for your methods and combined data for the methods they call.

Select this option	In this situation
Full Encapsulation	You collect data for your methods and the methods they call.

To set the recording level for your application, move the slider up or down. The selected option is highlighted in blue. The position of the slider indicates the trade-off between the amount of collected data and the impact on your application's performance.

The *Set Recording Level* dialog box is always initialized to the value you have last set. To see the settings in effect in the JProbe Analysis Engine, check the Session History (for details, see "Viewing the Session History" on page 445).

Note Unless the Data Recording is off, a new recording session always begins after you click **OK**, even if you did not change the level.

Running an Analysis Session

This procedure assumes that you have completed the following steps:

- created a basic configuration
- set performance options

Ideally, you should also set a current performance baseline and future performance expectations.

This section includes the following topics:

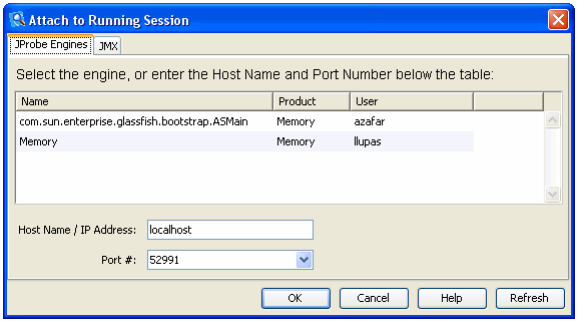
- Exercising Your Program
- Saving Baseline Snapshots
- Understanding Data Collection

Exercising Your Program

To exercise your program:

1. Start your program.
2. In JProbe, click **Attach to Session**  on the toolbar.

The **Attach to Running Session** dialog box appears.

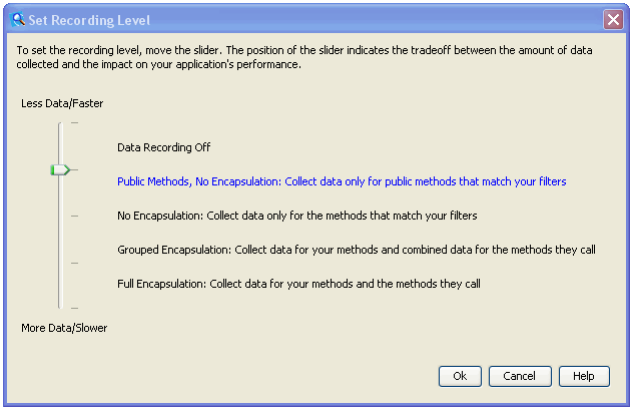


3 Select the appropriate engine from the list and click **OK** to connect JProbe to your application.

JProbe begins to collect data on your program and displays it in the *Runtime Summary* view.

4 In the *Runtime Summary* view, click **Set Recording Level** on the toolbar.

The *Set Recording Level* dialog box appears.


Set Recording Level
To set the recording level, move the slider. The position of the slider indicates the tradeoff between the amount of data collected and the impact on your application's performance.
Less Data/Faster
Data Recording Off
Public Methods, No Encapsulation: Collect data only for public methods that match your filters
No Encapsulation: Collect data only for the methods that match your filters
Grouped Encapsulation: Collect data for your methods and combined data for the methods they call
Full Encapsulation: Collect data for your methods and the methods they call
More Data/Slower
Ok
Cancel
Help

5 If data recording is off, turn it on by moving the slider down (for details, see "Setting the Recording Level" on page 236), and click **OK**.

Note Unless the Data Recording is off, a new recording session always begin after you click **OK**, even if you did not change the level.

6 Exercise your program.

7 If desired, take one or more snapshots by clicking **Take Snapshot** on the toolbar.

8 When the program has completed, click **Set Recording Level**, set the Data Recording to off, and click **OK**.

The data collected is saved in a temporary snapshot.

9 To end the session, click **Detach From Session** on the JProbe Console toolbar, then exit your program as you normally would.

The *Summary* view opens with the most recent performance snapshot displayed.

Saving Baseline Snapshots

You can save the snapshots you take during the exploratory session so that you can use them later on as a baseline. A baseline will help you to quantify improvements to yourself, your team members, quality assurance, product management, marketing, and other stakeholders. For more information, see "Measuring Performance Improvements" on page 258.

Note You may need to demonstrate improvements against an externally set benchmark. JProbe is a pre-production, performance analysis tool, not a benchmarking tool; your JVM may have performance characteristics that are very different when run with and without the profiling interface. In this case, you should use JProbe to set a baseline, find performance bottlenecks, fix the code, and verify the performance improvement against the baseline. To measure against the benchmark, rerun the fixed code without JProbe in the benchmarking environment and collect timing data.

The JVM you select and JVM options you set can influence how your program runs; your program will likely run differently under JVMs from Sun, IBM, or BEA because vendors optimize data and implement different garbage collection strategies in diverse ways. When running JProbe, your Performance analysis results will also differ across JVMs.

Due to JVM optimizations, even using the same JVM for multiple sessions may cause different results. You should run a few sessions under your selected JVM to allow the optimizations to occur. When you begin to get fairly consistent values, take your baseline snapshot. If the values continue to fluctuate significantly, elements of your development environment may be affecting the results. Consider switching to CPU time to avoid including time spent in program pauses or while waiting for I/O. For more information, see "Changing How You Measure Time" on page 257.

You save baseline performance snapshots in exactly the same way as you save any snapshot—you do not have to do anything special. For convenience, however, you may want to consider saving the baseline snapshots in a separate directory and/or with unique names. For example, you might create a Baseline directory and save the performance snapshots with names that reflect the sessions you ran. For details, see "Saving Snapshots" on page 448.

Understanding Data Collection

Before you begin identifying and investigating performance bottlenecks, you need to understand the data that is collected during the session. JProbe collects data for each method called by your program in terms of up to eleven performance metrics. Four of these metrics are collected, the rest are calculated. The basic metrics include all invocations of the method on all threads during the execution of your program.

Basic Metrics

The following table presents the basic metric types.

Basic Metrics	Description
Number of Calls	Number of times the method was invoked
Method Time	Amount of time spent executing the method, *excluding* time spent in methods it called
Number of Catches	Number of times the method catches an exception
Number of Exception Exits	Number of times the method exits by exception

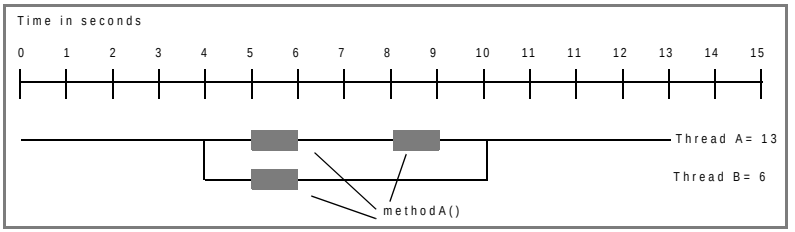
Catch and Exception Counts

JProbe collects and displays method-level information on how many exceptions are caught by instrumented methods, and how many times those methods were exited via the exception mechanism. This information displays in several Performance views including the Methods, Method Detail, Snapshot Difference, and Performance Report.

Metrics and Threaded Programs

In multithreaded programs, data for a method includes all invocations of that method on all threads. For example, if thread A invokes `methodA()` four times and thread B invokes the same method six times, the Number of Calls metric for `methodA()` is ten (4+6=10). While it may seem obvious, totaling time on all threads can lead to times for the program that do not match the time you subjectively experienced. Consider the following example where `methodA()` is executed by two threads and time is measured as elapsed time in seconds:

Tip You can recalculate the data to represent your main thread of execution only by selecting the thread node and isolating its call tree. See "Focusing on Threads and Methods in the Call Graph and Call Tree" on page 249.



Your experience of this program is 13 seconds. The total cumulative time for the program, however, is reported as 19 seconds (13 seconds on thread A *plus* 6 seconds on thread B). The difference is the cumulative time contributed by thread B. The above example shows that method calls on both threads are accumulated—`methodA()` takes one second to execute and is invoked a total of three times, so the method time for `methodA()` is 3 seconds (2 seconds from thread A + 1 second from thread B). The total cumulative time for the program simply adds all the method times together. The solution is to isolate the thread of interest (in this example thread A) in the Call Graph.

Metrics and Data Collection Filters

The following table describes the data collection filters that are available:

Detail	Description
Method Level	Use the Method Level detail filter for your own code. The data collected includes metrics for all parts of the method that were executed.
Encapsulated	Use the Encapsulated detail level to hide details on code that you are not interested in analyzing. By default, data on system classes, third-party packages, and frameworks is encapsulated. For more information, see “Encapsulating Data on Methods” on page 250.
Line Level	Use the Line Level detail filter *only after* you identify hotspots, and then only when you need a line-by-line breakdown of how time is spend in a method. Collecting line-level detail takes more time and produces larger snapshots. For more information, see “Collecting Data on Lines of Code” on page 255.
Native	By default, time spent in `native` methods is attributed to the method that calls it. Use the Native filter to identify `native` methods in your code. The time for these methods is then tracked separately from its calling method. You do not need to use Native filters if your JVM is Java 6 or newer. Native filters are only required for older JVMs, where `native` methods cannot be instrumented directly. Also, you do not need to identify `native` methods in third-party packages and frameworks. **Note** Do not use Method Level filters for any native methods. If you do use Method Level filters for any native methods JProbe warns you and suggests a native filter. If you ignore the warning, the data JProbe collects may be incorrect.

Caution Filter order is important. When two entries refer to the same code, the entry lower in the table dominates.

To set data collection filters:

1 Click **Tools > Create/Edit Settings** to open the JProbe Create/Edit Settings dialog box.

2 Click **Edit** to change the settings.

3 If the method is already listed in the table, select an action by clicking the arrow in the Action column. Otherwise, add a new row for the method. Arrange the rows in the order in which you want the filters applied.

4 Click **Save,** then **Close**.

Investigating Your Results

This section includes the following topics:

- Identifying Performance Bottlenecks
- Investigating Performance Bottlenecks
- Narrowing Your Investigation
- Measuring Performance Improvements
- Identifying Deadlocks
- Investigating Performance Deadlocks

Identifying Performance Bottlenecks

Your goal is to locate your most expensive methods in terms of one or more metrics. As your program runs, the Heap Usage Chart acts as a progress bar, displaying memory use over time and reporting on garbage collections. By default, JProbe takes a performance snapshot when the recording level changes, or when data collection ends. You can use either the Call Graph or Call Tree, or the method list under either Call view to find the expensive methods — whichever is your preference or works best with your application.

Three approaches to identifying potential bottlenecks include:

- Finding Hotspots
- Finding Hotspots in the Critical Path
- Finding Hotspots by Navigating the Call Graph

Finding Hotspots

The method list below the Call Graph/Tree is the simplest place to start. You can filter the list to show only your methods first and remove encapsulated, third-party methods

from the view. You can then sort the list in terms of each of the metrics. Your most expensive methods appear at the top of the list (or the bottom if sorted in ascending order). Compare the results against your expectations. If you are analyzing a standalone application, see "Finding Hotspots by Navigating the Call Graph" on page 247.

To find the most expensive methods using the Call Graph method list:

1. If the snapshot is not already open, in the Snapshots Navigator double-click the performance snapshot you want to analyze.

   The *Summary* view opens.

2. Double-click the method of interest.

   The *Methods* view opens with the selected method as the focus method.

3. Enter your package name in the Filter Methods field.

   The list displays only the methods from your package.

4. If the list is not sorted by Method Time, click the **Method Time** column heading twice.

   Your most expensive methods in terms of method time appear at the top.

5. Assess each of the top ten methods, looking for ones that take more time than you expected.

6. After you identify one or more hotspots, go to "Investigating Performance Bottlenecks" on page 248.

Finding Hotspots in the Critical Path

The simple method described above has two drawbacks. One is that the hotspots may only be "hot" on some call paths. This is not apparent from the method list alone, but you can see it in the Call Graph. The other drawback is that the hotspot may include data that is not relevant, if it was called along different paths. The Call Graph can help by showing the critical path, and by allowing you to isolate or prune to focus only on relevant data.

By default, the nodes in the graph are colored to show the most expensive methods in your program in terms of cumulative time. You can easily identify the *critical path* of your program by following the brightest nodes in the graph. The critical path represents the main path of execution for the session or sessions you ran. A bottleneck in your critical path has the most impact on your customer's perception of the performance of your application.

Threads are excluded from the Call Graph by default, because the vast majority of threads are created by the application server. You have no control over these threads, so you can safely hide the data associated with them. What you do see are your application's methods. Your data set includes only the call trees initiated by the methods that started your Java EE application.

In the Call Graph, you can include the threads in the data set. Including threads allows you to analyze their call trees collectively and individually. For example, if you find an expensive method invoked by two threads, you can select one thread and isolate its call tree. The data changes to reflect only the single thread. If the method is still a hotspot, the selected thread is probably the cause. To confirm your assumption, you can reset the graph and select the other thread.

Regardless of the type of application you are analyzing, your results would be clearer in the Call Graph if you used the data collection filters. Your own methods stand out if you collected method-level detail on your own package, and left all other code in the encapsulated state. Encapsulated data is displayed in the Call Graph as locked nodes, (that is, there is a lock icon on the node) or nodes with an asterisk (Encapsulated using the Grouped option).

To identify hotspots along the critical path:

1. In the Snapshots Navigator, double-click the Performance snapshot that you want to analyze.

   The Summary view opens, providing you with a list of potential hotspots and suggesting the top five focus methods with which you can continue your Performance analysis.

   Note When JDBC data is recorded in the snapshot, the **Methods** tab is displayed on the foreground and the **JDBC** tab is displayed on the background.

2. In the **Methods** tab, select the *Elapsed Time* or the *CPU Time* option to display the suggested methods based on elapsed time data or on CPU time data (provided that the latter is available).

3. Select a method in the list, right-click, and select **Investigate**.

   The Methods view opens.

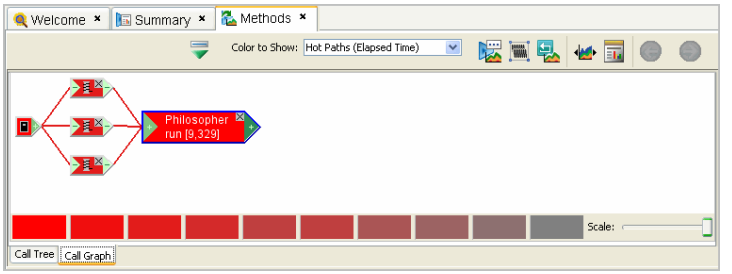
4. Ensure that the Call Graph is set to sort methods by **Elapsed Cumulative Time** and the *Color to Show* metric is set to **Hot Paths (Elapsed Time)**.

5. If you are analyzing a Java EE application, you can focus your investigation by isolating the tree that represents your application. Find the entry point to your application (for example, a `doGet()` or `doPut()` method), select a method, right-click, and select **Isolate Selected Node**.

The data set is recalculated to include only the methods called by your application.

6 To zoom out so that you can see the critical path more clearly, move the **Scale** slider to the left.

Cumulative time includes the time spent in child methods. Therefore, your Call Graph is brighter on the left side where your root nodes appear. You should also see a bright path of nodes through the Call Graph from left to right. This is your critical path.



7 Follow the critical path from left to right, expanding the call tree as necessary, and select the last bright node that represents one of your methods. For details, see "Finding Hotspots by Navigating the Call Graph" on page 247.

8 Zoom back in on the selected node by sliding the Scale button to the right. Position your mouse cursor over the selected node in the graph.

A tooltip shows the method name and signature, the cumulative time, and the number of calls to the method.

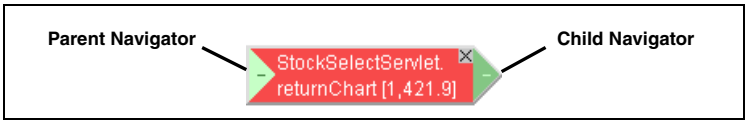
9 If a critical path does not stand out, do one of the following:

- Display a higher number of nodes by clicking **Show More Nodes**. You can add nodes along the critical path until you reach the root.
- If there is little difference among the node colors in the graph, try switching the color scale to Linear Mapping. For details, see "Changing the Scale Gradation" on page 323.
- Select **Hot Methods (Elapsed Time)** from the *Color to Show* list. The graph is repainted to show the methods that take the longest to execute, excluding time spent in child methods. Select a bright node representing one of your methods to investigate.

10 After you identify one or more hotspots, go to "Identifying Performance Bottlenecks" on page 243.

Finding Hotspots by Navigating the Call Graph

When a method's cumulative time is high, but the method time itself is relatively low, it means that the method is calling one or more expensive methods. The Call Graph shows you the calling relationships in a directed graph. By navigating the graph, you can find the hotspots in the call tree. To navigate the graph, you use the navigators on the node in the graph. The navigator on the left, called the parent navigator, is for traversing the methods that call the selected method. The navigator on the right, called the child navigator, is for traversing the methods called by the selected method.



In many cases, the hotspot will be represented as an encapsulated node. If the real hotspot is from someone else's code, you still need to focus on your own calling method. Try to reduce calls to an expensive method, or try to find more efficient ways to call it.

To identify a hotspot by navigating the Call Graph:

1. Ensure that the *Color to Show* metric is set to show **Hot methods (Elapsed Time** or **CPU Time)**. Adjust if necessary.
2. Select the expensive method node in the Call Graph. If the node is not visible, right-click the method and select **Fit Method Into Graph** to show it in the Call Graph.
3. With the method node highlighted in the Call Graph, click the child navigator.

   Note In some cases, a hotspot is a small, frequently-called method, such as an accessor. It is unlikely that this is your performance bottleneck, but it may be hiding other hotspots.
4. If all the expensive child methods are encapsulated, the hotspot remains the calling method. Usually, they will be methods from third-party packages and frameworks, but if an expensive, encapsulated node represents your own code, you should rerun the session with a method-level data collection filter set on the method.
5. If one of the child methods is very bright and has method-level detail, select the child node. Expand its children (if it has any). Repeat until you find a bright (unencapsulated) leaf node or a bright node with dull child nodes; the bright node is the hotspot.

Using the Call Tree

The Call Graph is useful for showing paths, but because the nodes are large, it is not very compact. Sometimes the Call Tree is easier to navigate than the Call Graph.

The Call Tree displays the CCT (see "How JProbe Builds a Calling Context Tree" on page 223) as a tree table that you can navigate by expanding and collapsing the nodes. To use the Call Tree, open the Methods view, and click the **Call Tree** tab.

Investigating Performance Bottlenecks

Now that you have identified a hotspot in your program, you need to determine if it is really a candidate for improvement, and, if so, what degree of control you have over the algorithm. You can drill down to more information using the Method Detail view.

The Method Detail view displays details on the method in the call path for which you are currently viewing data. For more information, see "Exploring the Method Detail View" on page 325.

To investigate a performance bottleneck:

1. In the Call Graph, select the hotspot method you identified.

   The method is highlighted in the list and in the graph (if visible).

2. Right-click the selected method and select **Open Method Detail View**.

   The Method Detail view opens with the selected method displayed in the Current Method field.

3. Evaluate how the Current method is called by reviewing the information in the Edges From Parents area.

4. Evaluate the cost of the methods called by the Current method by reviewing the information in the Edges To Children area.

5. You can navigate by using the navigation arrows and by double-clicking any method.

   The Method Detail changes to show the new selected method in the Method area. The Parents and Children tables now reflect data in terms of this method.

   Note To return to the last Current Method, select the back arrow, or click the forward arrows to navigate to the next method in our method viewing history.

6. Evaluate the code for the method itself by clicking  **Show Source**.

   The Source view opens with the source code containing your method.

Next Steps

Consider your alternatives for improving the speed of the algorithm contained in this method. You may be able to modify the algorithm to run faster, cache results for future use, or change the way you call other methods. Rerun the session with the modified code using the same JPL settings file and take a performance snapshot. You can then use the Snapshot Difference view to assess your modifications. For more information, see “Measuring Performance Improvements” on page 258.

By improving performance bottlenecks in the critical path, you will achieve visible improvements in the runtime of your program. If there is nothing you can do with this method, repeat the procedures to find the next problem method.

Narrowing Your Investigation

Up to now, you have used the default data collection options to investigate potential performance bottlenecks. You may want to focus on a single call tree, change how you measure time, collect line-level data for a few methods, or change other data collection filters.

You can narrow your investigation in the following ways:

- Focusing on Threads and Methods in the Call Graph and Call Tree
- Encapsulating Data on Methods
- Collecting Data on Lines of Code
- Collecting Data Using Triggers
- Changing How You Measure Time

Focusing on Threads and Methods in the Call Graph and Call Tree

You can narrow your investigation by isolating a single thread or method subtree. All nodes that are not part of the selected node’s subtree are removed from the graph and tree, and the method list metric data is recalculated to include only data for the isolated subtree. You can also prune threads and method subtrees from the graph or tree. In this way you can remove their nodes from the Call Graph and Call Tree—and their contributions to cumulative time from the method list—and focus on the performance of the rest of the program. For more information, see “Editing the Data Set” on page 318.

Strategies for using the isolate and prune features include:

- Focusing on a Method

- Removing Threads and Method Subtrees

Focusing on a Method

You can select any method node and isolate its subtree. The data will reflect only this method plus any methods called by this method. Focusing on a method is particularly useful for Java EE applications, where you want to focus on your application or a small part of a large application. For example, you might focus on a `doGet()` or `doPut()` method. For more information, see “Isolating Subtrees” on page 319.

Removing Threads and Method Subtrees

You may decide, while investigating a snapshot, that you no longer want to include the data for a particular thread or method. For example, you may decide that there is nothing you can do about the performance of the Java class loader, and you want to assess the performance of the program without this method and the methods it calls. You can select the `java.lang.ClassLoader` method and prune its subtree. Other typical data that you may want to prune from the data set includes time spent waiting (calls to `Object.wait()`) and I/O time (calls to methods in `java.io.*)`. For more information, see “Isolating Subtrees” on page 319.

Encapsulating Data on Methods

Encapsulation means to hide details behind a well-defined interface. When one of the encapsulation options is used, Performance tracks the call into the method and then just accumulates data until the method calls out to a non-encapsulated method.The encapsulated method is a “black box,” hiding details about calls to other encapsulated methods. In general, encapsulation improves session runtime performance and reduces the amount of data you need to assess. You will see different results depending upon the type of encapsulation that you run with your session.

After running a session and investigating hotspots you will identify methods that are working well. In subsequent sessions, you can encapsulate data on these methods so that you can focus on areas of interest.

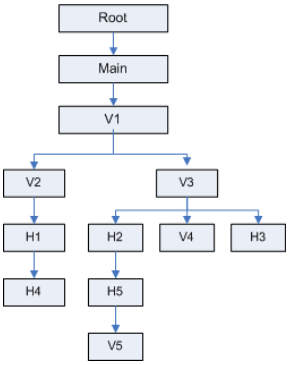
Filtering Encapsulated Methods

Performance filters in the JProbe settings tell JProbe which methods are considered encapsulated. By setting filters you can include only the methods that you want to view in the Call Graph. In addition, by using the Encapsulated action, Performance controls how the filtered methods should be treated. For more information, see “Refining Data Collection Filters” on page 422.

Encapsulation Types

There are three levels to choose from to collect encapsulated data: Full, Grouped (default), and None. Each setting provides a different level of abstraction for viewing method information. Encapsulating data assists you in focusing on hotspots and data that is important to you.

To begin investigating the encapsulated options, let's examine a call tree representation of methods captured in your application. The parent methods and their encapsulated children method are displayed in a hierarchical view.



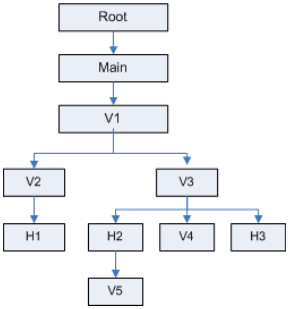
Unfiltered Call Tree

During filtering, the call tree is mostly preserved, but there may be some methods that are not exposed. For example, V3 may have called a method that called another method that eventually called V5.

Full Option

Tip The Full option requires a lot of resources during the collection of data impacting the amount of time it takes to run a session.

Choosing the Full encapsulated option provides the most comprehensive way of viewing method information. The structure of this call tree is most like the call tree representation of your application data. The data shows only the first encapsulated method; all other information about encapsulated methods is removed from the call tree.



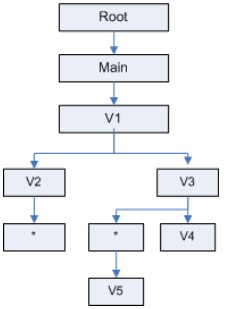
Encapsulation using Full option

Notice in the "Unfiltered Call Tree" on page 251 that the nodes H4 and H5 are displayed in the call tree, but when using the Full encapsulated option, the nodes H4 and H5 are encapsulated and not shown. The time spent in these nodes is directly attributed to their parent methods.

Grouped Option

Tip In comparison to the Full encapsulation option, the Grouped option does not substantially impact on performance.

The Grouped encapsulated option aggregates all of the encapsulated methods called by a particular method into a single method or node. This type of encapsulation simplifies the information that is displayed in the Call Graph because the methods are not shown by name, but rather, by an asterisk (*), and there may be fewer nodes. Nodes marked with an asterisk appear in the Call Graph, but their information does not display in the list of methods below the Call Graph.

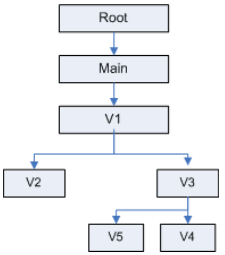


Encapsulation using Grouped option

If you compare the grouped call tree with the "Unfiltered Call Tree" on page 251, you can notice that all of the children methods (H*) in the grouped diagram are represented by an asterisk placeholder. The placeholders represent all encapsulated children of the calling method. For example, in the diagram, the left-most asterisk encapsulated all of the method time for H2, H5 and H3.

None Option

The encapsulated type None does not display any encapsulated methods in the Call Graph.



Encapsulation using None option

If you compare the None diagram with the "Unfiltered Call Tree" on page 251, you can notice in the None diagram that none of the H* methods display in the Call Graph. In fact, when you view the diagram, there is no indication that these methods were invoked during the session. All of the attributes of the children methods are directly attributed to the calling method. For example, the time spent in the method H1 and H4 is attributed to V2's method time and the time for H2, H5, and H3 is attributed to V3's method time.

To encapsulate data on methods in your application:

1. In the JProbe Console, select a configuration from the Configurations Navigator.

   The **<Configuration_File_Name.jpl>** tab appears in the View area.

2. Click **Edit**.

   The **<Configuration_File_Name.jpl>** tab is placed in editing mode.

3. On the **Analysis Type** tab, select the Performance Analysis check box.

4. On the **Automation** tab, move the slider to select one of the options available in the *Initial Recording at JVM Start* section:

   - Data Recording Off
   - Public Methods, No Encapsulation
   - No Encapsulation
   - Grouped Encapsulation
   - Full Encapsulation

   Note Unless you need to track things from the start of the JVM, it is recommended to set the initial recording to *Data Recording Off* and use triggers to start recording at a relevant time.

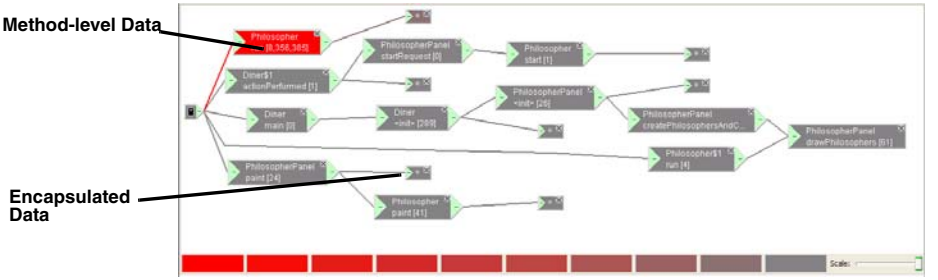
5. Click **Save**.

6. Start a Performance analysis session.

7. To modify the recording level during the analysis session, click **Set Recording Level**  on the *Runtime Summary* view's toolbar.

   The *Set Recording Level* dialog box appears.

8. Select one of the encapsulation options and click **OK**.



Collecting Data on Lines of Code

By default, Jprobe accumulates data for the methods in your program. This level of detail is sufficient for finding performance bottlenecks. After you find a bottleneck, however, you may want more information on the methods contributing to the bottleneck. You can rerun the session with a filter on the problem method to track time for each line of code in that method. The metrics collected are similar to the method-level metrics, but are adjusted to represent lines of code. Cumulative and Averages metrics are also for each line. To see line data, you need to open the Source view.

Note Collecting line data is time-consuming and will impact session performance and the accuracy of other method data. Use Line Level detail only after you have found one or two methods that contribute to a performance bottleneck. To view line-level data for methods, you must have the corresponding source code.

With line-level detail, JProbe collects data for each line listed in your compiled program's *line data table*. As you may know, the line table matches up the compiled bytecode with the actual line of Java code in your uncompiled program. Sometimes the match is imprecise because the bytecode represents more than one line of code. For the purpose of debugging performance bottlenecks, however, the line data table provides enough accuracy to point you in the right direction. You can try to improve the line data table by compiling your program with the Java compiler's `-g` option.

To obtain line-level information, your code needs to be compiled in debug mode. The following example demonstrates how to modify an Ant compile target to turn on debug mode.

Assuming that your compile target reads as follows:

```
<target name="compile" depends="init"
    description="compile the source " >
```

```
    <!-- Compile code from ${src} and ${src.test} into ${build} --
>
    <javac destdir="${build}" >
        <src path="${src}" />
        <src path="${src.test}" />
        <classpath refid="classpath"/>
    </javac>
</target>
```

Create the debug compile target with the following modifications (in bold):

```
<target name="compile.debug" depends="init"
    description="compile the source " >
    <!-- Compile code from ${src} and ${src.test} into ${build} --
>
    <javac debug="on" destdir="${build}" >
        <src path="${src}" />
        <src path="${src.test}" />
        <classpath refid="classpath"/>
    </javac>
</target>
```

To collect line data:

1. Open your configuration file in editing mode.
2. In the **Analysis Type** tab, click the **Filters** tab.
3. If the method is already listed, select **Line** from the list in the Action column. Otherwise, add a new row for the method. Arrange the rows in the order in which you want the filters applied.

   Caution Filter order is important. When two entries refer to the same code, the entry lower in the table takes priority.

4. To view line data, select the method in the Call Graph, Call Tree, or Method List, right-click, and select **Show Source**.

   The Source view opens with the source file containing the method code displayed. Line data appears in the columns to the left of the method.

Collecting Data Using Triggers

Triggers allow for automated data collection and for precise control over what is recorded. They are also useful when you are investigating a program that executes very quickly—faster than you could take a snapshot manually. You can take a snapshot automatically with a trigger. For more information, see “Setting Triggers” on page 425.

Changing How You Measure Time

You can measure time by either Elapsed Time (default) or CPU Time. Review the descriptions to determine which one suits your investigation. You select the time measurement in the *Create/Edit Settings* dialog box, on the **General** tab.

Note JProbe is a performance analysis tool, not a benchmarking tool. Times taken while your program is being analyzed cannot be compared to the performance of your program running alone.

Elapsed Time

Use elapsed time to get a sense of how long your program takes to execute in real time. Elapsed time includes pauses, such as waiting for I/O, sleeps, and any other pauses in the normal execution of your program. Pauses may differ from run to run, so you should be careful when comparing snapshots using elapsed time.

Elapsed times can also be affected by the system resources available to your program, such as processor speed, other running applications, available memory, and so on. Within bounds, the more resources, the faster your program will run. Obviously, this means that your program's performance will differ from one computer to another, and across platforms.

CPU Time

Use CPU time when you want to separate the performance of your code from how your runtime environment handles your program. CPU time does not include program pauses, such as I/O, and is not affected by system resources (with the exception of the timing resolution provided by the operating system itself). CPU Time is often more expensive to collect than Elapsed Time, so your program may run noticeably slower under JProbe when collecting CPU Time as compared to Elapsed Time.

Note Some operating systems do not offer CPU timing. In these cases, JProbe automatically uses elapsed time.

CPU time measures the number of processor cycles spent executing code in all threads. The resolution of the CPU timers is determined by the operating system, and is often much coarser than the resolution of elapsed time. When a resolution is comparatively coarse, it is entirely possible that a very small method, and certainly single lines of code, will execute within a single tick of the timer. As JProbe can only measure at the unit of

resolution provided by your operating system, time cannot be reported as fractions of this unit. Instead, the time for these small methods/lines of code will be reported as zero.

Measuring Performance Improvements

After you solve a performance bottleneck, you can measure how much of a performance improvement you gained. For example, you probably want to know if the algorithm itself is faster. You may also want to understand what impact the fixed algorithm has on the entire program.

Note To ensure you are comparing similar data, use the same settings file for both sessions. Remember to run the improved code a few times to allow optimizations to occur.

The tool for measuring performance improvements is the Snapshot Difference view. For details, see "Exploring the Performance Difference View" on page 338.

There are two approaches to measuring performance improvements:

- Measuring Improvements in a Modified Algorithm
- Assessing Overall Program Performance

Measuring Improvements in a Modified Algorithm

You can compare the performance snapshot for the modified code with the performance snapshot taken of the original code. The result will be a method-by-method summary of the increase or decrease in method times/objects. Decreases represent performance improvements and are shown as negative numbers.

Note The JVM you use can influence the performance of your program over time as optimizations are made. If you ran the original session a few times before taking the baseline snapshot, you should do the same with your modified code.

To measure the performance improvement for an algorithm:

1. Run the modified code using the original JPL settings file and take a performance snapshot. For details, see "Running an Analysis Session" on page 237.
2. In the Snapshots Navigator, load the baseline snapshot that contains data on the original code. For details, see "Saving Baseline Snapshots" on page 239.
3. Right-click the Performance snapshot containing data on the original code and select **Snapshot Differencing**.

The Performance Difference dialog appears.

4 In the Baseline Snapshot drop-down list, choose the snapshot with the modified code.

The Snapshot to Compare drop-down list already displays the snapshot with the original code.

5 Click **OK**.

The Performance Difference view opens.

6 Find the modified method in the table and read the value in the Cumulative Time column.

7 If you need more detail, select **Tools > Options**, choose **Performance** from the navigation tree, and from **Time Unit** select a smaller unit of time, such as microseconds or nanoseconds.

8 If the value is negative, it represents a reduction in time spent in the method, which translates to a performance improvement. You have succeeded in improving the algorithm.

9 If the value is positive, the modified code runs slower than the original code. You will probably want to try a different algorithm.

10 Close the Snapshot Difference view.

Assessing Overall Program Performance

When modifying an algorithm, you often have to make changes to other parts of your program. In this case, you need to assess the trade-offs of those modifications. You may have improved the algorithm itself, but added other code that impacts the runtime of the program. Alternatively, your algorithm itself may run more slowly, but changes to code elsewhere improve overall program performance.

The overall program performance is the Cumulative Time for the entry point for your application. For Java EE applications, you need to find the entry point to your application. In Servlets, for example, you would focus on the `doGet()` and `doPost()` methods. For standalone applications, you can look at the thread that runs your main method. You cannot, however, see overall program performance for all threads at once.

To measure the program performance:

1 Run the modified code using the original JPL settings file and take a performance snapshot. For details, see “Running an Analysis Session” on page 237.

2 In the Snapshots Navigator, load the performance snapshot that contains data on the original code. For details, see “Loading Saved Snapshots” on page 449.

3 Right-click the Performance snapshot containing data on the original code and select **Snapshot Differencing**.

The Performance Difference dialog appears.

4 In the Baseline Snapshot drop-down list, choose the snapshot with the modified code.

The Snapshot to Compare drop-down list already displays the snapshot with the original code.

5 Click **OK**.

The Performance Difference view opens.

6 Use the method filter and column sorting to find the entry method for your application.

The cumulative time for the method is the overall performance of your application.

7 If the overall performance has improved, your modifications have been successful. Determine if the percentage reduction in time/objects is enough, or if you need to attempt further improvements.

8 If the overall performance is the same or has deteriorated, investigate individual methods. Note where you reduced time/objects and where you increased them. Determine if you can make further improvements, or if you should revert to the original code.

9 Close the **Performance Difference** view.

Identifying Deadlocks

Performance can report thread deadlocks in your Java SE and Java EE applications.

This section includes the following topics:

- Understanding Deadlocks
- Finding Deadlocks

Understanding Deadlocks

A deadlock is a situation in which a cycle of threads is blocked because they are waiting for access to a resource that is held by another thread in the cycle and so can never be freed. The program can never terminate because the threads are blocked indefinitely. With standalone applications, you experience a deadlock as a *hang*, that is, the program becomes unresponsive and never ends. Deadlocks in Java EE applications may result in incomplete transactions or a high number of waiters in a thread pool.

Note In the Diner example you can see a program that deadlocks. Click **Help > Content** and **Performance Tutorials >Diner**.

This behavior results from improper use of the `synchronized` keyword to manage thread interaction with specific objects. The `synchronized` keyword ensures that only one thread at a time is permitted to execute a given block of code. A thread must therefore have exclusive access to the class or variable before it can proceed. When it accesses the object, the thread locks the object, and the lock causes other threads that want to access that object to block until the first thread releases the lock.

Since this is the case, by using the `synchronized` keyword, you can easily be caught in a situation where two threads are waiting for each other to do something.

A classic example for a deadlock situation is as follows:

```
class Deadlocker
{
int field_1;    private Object lock_1 = new int[1];
int field_2;    private Object lock_2 = new int[1];
public void method1( int value )
    {       synchronized( lock_1 )
        {       synchronized( lock_2 )
            {
                field_1 = 0;
                field_2 = 0;
            }
        }
    }

public void method2( int value )
    {       synchronized( lock_2 )
        {       synchronized( lock_1 )
            {
                field_1 = 0;
                field_2 = 0;
```

```
            }
        }
    }
}
```

Consider this situation:

- One thread (Thread A) calls `method1()`.
- It then synchronizes on `lock_1`, but may be preempted at that point.
- The preemption allows another thread (Thread B) to execute.
- Thread B calls `method2()`.
- It then acquires `lock_2`, and moves on to acquire `lock_1`, but cannot because Thread A has `lock_1`.
- Thread B is now blocked, waiting for `lock_1` to become available.
- Thread A can now resume, and tries to acquire `lock_2`. It cannot because Thread B has acquired the lock already.
- Thread A and Thread B are blocked. The program deadlocks.

Of course, most deadlocks are not so obvious simply from reading the source code, especially if you have a large multithreaded program. Performance can identify deadlocks and point out where in the source code these deadlocks originate so that you can fix them.

Hangs That Are Not Deadlocks

In some cases, your program may hang even though it does not contain a deadlock situation. The difference between a deadlock and other kinds of hangs is that, in a deadlock situation, the threads can *never* move forward, whereas in a hang, the threads *will not* move forward. The most common example of a hang is when a thread has called a `wait()` method, but no thread ever calls the `notify()` method to wake it up.

Consider a program that has a producer thread and a consumer thread:

```
while(!done)
{
    Object item = produceItem();
    synchronized (queue_)
    {
        queue_.add(item);
        queue_.notify();
    }
```

```
    ...
}

while(!done)
{
    synchronized (queue_)
    {
        if (queue_.isEmpty())
        {
            queue_.wait();
        }
        item = queue_.remove();
    }
    useItem(item);
    ...
}
```

Suppose `produceItem()` throws an exception while the consumer thread is executing `queue_.wait()`, and the producer thread terminates. The consumer thread is never notified, and the program hangs. It is possible, however, that the program could continue in this situation. For example, the main thread might notice the terminated producer thread and create a new producer, in which case, the program would continue.

A hang can also be caused by problematic thread priority. For example, the thread priority might keep a lower priority thread from ever running, possibly while the lower priority thread holds a resource required by a higher priority thread (called a *priority inversion*).

Performance does not detect these hangs as deadlocks.

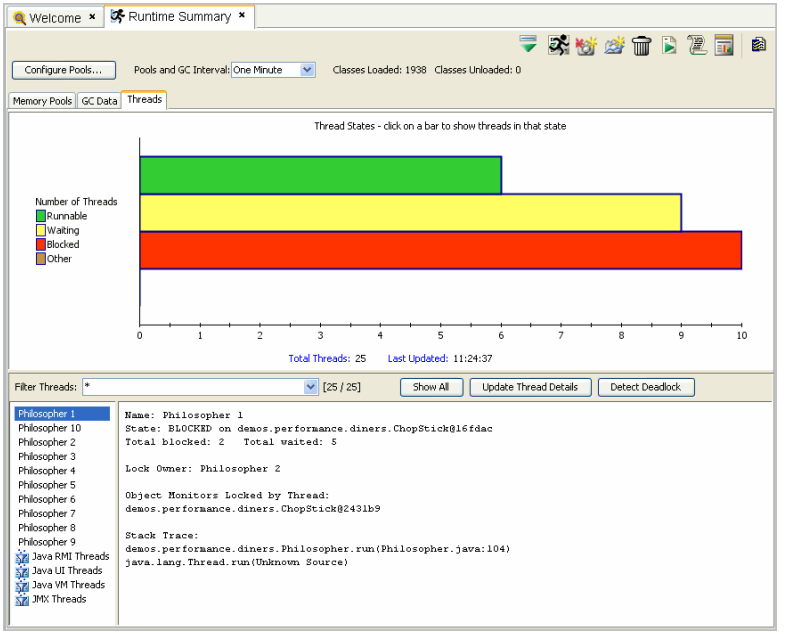
Finding Deadlocks

JProbe can detect deadlocks on demand.

To detect deadlocks while running a Performance analysis:

1. In JProbe, connect to a running session of your program configured to execute a Performance analysis.
2. In the Runtime Summary view, click the **Threads** tab.
3. Click **Detect Deadlock**.

   If deadlocks are detected, the list of threads in the lower pane is updated to show only the threads involved in the deadlock. If no deadlock is detected, a message is displayed to that effect.



Investigating Performance Deadlocks

After you execute the procedure described in section Finding Deadlocks, the lower pane in the **Threads** tab is updated to display only the threads that caused the deadlock(s). These threads are displayed in the graph as *Blocked*.

To view details about a particular thread involved in the deadlock situation:

- Click the thread name in the threads list.

  The following information is displayed in the right side of the pane:

  - The thread name and ID.
  - The thread state (BLOCKED) and the object on which the thread is locked on.
  - The number of times the blocked thread tried to enter or re-enter a monitor (that is, the number of times the thread has gone into the BLOCKED state), and the number of times the thread waited for notification (that is, the number of times the thread has gone into the WAITING or TIMED_WAITING state).
  - The name and ID of the thread locking the selected thread.
  - The stack trace and source code line of the method that caused the deadlock.

11

Learning More About Performance Options

This chapter presents how to fine-tune the Performance analysis tool and how to analyze a heap dump.

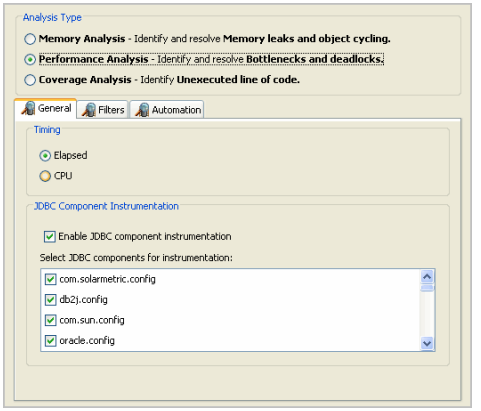
This chapter contains the following sections:

Analysis Options for Performance 266

Analysis Options for Performance

You can configure the following data collection settings for your Performance analysis:

- Timing Settings for Performance
- JDBC Component Instrumentation
- Filters for Performance
- Automation Settings for Performance
- Triggers for Performance



Timing Settings for Performance

You can select the *Timing* settings for your Performance analysis in the **General** tab of the configuration wizard.

By default, JProbe calculates method execution times using elapsed time. Elapsed time includes program pauses, such as wait times for resources. By including pauses, you gain an understanding of the performance of your program from a user's perspective. You may also find it helpful to judge the performance of your code separately from the performance of the environment in which it runs. In this case, you can omit program pauses by selecting CPU time. CPU timing is not available on all operating systems.

Timing	Description
Elapsed	Default. Elapsed time includes program pauses, such as time spent waiting for I/O.
CPU	CPU time ignores pauses.

JDBC Component Instrumentation

You can select the *JDBC Component Instrumentation* settings for your Performance analysis in the **General** tab of the configuration wizard. By default, all JDBC components identified for instrumentation are enabled.

Database component instrumentation is offered by identifying driver classes for instrumentation. This feature allows you to distinguish between the time spent in their code and time spent working with the database component. Common database drivers and those that ship with supported application servers are recognized by default.

When launching a Performance analysis with JDBC component instrumentation enabled, JProbe instruments the JDBC driver classes and treats them as a component boundary. The methods invoked from the JDBC classes are presented in the call tree as JDBC nodes, with additional data.

Filters for Performance

You can select *Data Collection Filters* for your Performance analysis in the **Filters** tab of the configuration wizard.

Filters help you to focus on those aspects of the program that are important to you. Your Data Collection Filters table may already contain a set of default filters. The default filters are based on the description of your code that you provided (see the **My Code** tab in the Create/Edit Settings dialog box). You can use these filters or edit them.

Filters are defined in the same way for all the analysis tools, however the filter actions differ among the analysis tools. This section summarizes the filter actions for a Performance analysis. To learn how to create filters, see Refining Data Collection Filters.

Filter Action	Description
Method	Default. Collects method-level data on the specified code. Select this action for code that you are interested in analyzing.
Encapsulated	Encapsulates data about the specified code. Select this action when you do not care about the performance of the specified code or you do not have control of the source code. **Note** How JProbe collects and presents data for encapsulated methods is controlled by an encapsulation policy. For more information, see "Encapsulation Policy" on page 269.
Line	Collects line-level data for each line of code in the specified package, class, or method. Select this action for a method that you know (from a previous method-level analysis) is contributing to your bottleneck. It will help you to pinpoint problem lines of code.
Native	Encapsulates data when your application calls the specified native methods. Select this action for all native methods. If you do not select this action, some of the time spent in a native method may be attributed to its calling method. You do not need to use Native filters if your JVM is Java 6 or newer. Native filters are only required for older JVMs, where `native` methods cannot be instrumented directly. **Note** Avoid assigning this action to non-native methods. If you do, the resulting data may be incorrect.

Automation Settings for Performance

You can select initial recording and encapsulation options for your Performance analysis in the **Automation** tab of the configuration wizard.

Initial Recording Options

The following *Initial Recording at JVM Start* options are available for your Performance analysis:

- Data Recording Off
- Public Method, No Encapsulation

- No Encapsulation
- Grouped Encapsulation
- Full Encapsulation

Encapsulation Policy

An encapsulation policy determines how data for methods marked “encapsulated” is collected and how that data is attributed. Encapsulation can be specified as a Recording Parameter when you are adding a trigger. For more information, see “Triggers for Performance” on page 271.

After you finish running the session, you will examine the resulting snapshot of data in the Methods view. The encapsulation policy you selected affects how some nodes are displayed in the graph. The selected policy is displayed in the status bar of the Methods view. For more information, see “Exploring the Methods View” on page 297.

To help distinguish among the available policies, consider the following call sequence: A non-encapsulated method (`methodA`) calls out to an encapsulated method (`methodX`), which in turn calls other encapsulated methods (`methodY` and `methodZ`). `methodZ` then calls another non-encapsulated method (`methodB`).

Policy	Description	Call Sequence Example	Advantage/ Disadvantage
None	JProbe aggregates data for the non-encapsulated calling method plus the encapsulated methods and attributes it to the calling method. In the Call Graph, the encapsulated methods are not displayed.	The data for `methodA`, `methodX`, `methodY`, and `methodZ` is aggregated and attributed to `methodA`. In the Call Graph, `methodA` calls `methodB`.	With this policy, data collection is the fastest. It produces a simplified Call Graph, but the data for your calling method is inflated with performance data for methods that you may have no control over.

Policy	Description	Call Sequence Example	Advantage/ Disadvantage
Grouped	JProbe does not collect data for encapsulated methods. In the Call Graph, calls to one or more encapsulated methods are represented by a single link to a node containing only an asterisk (*).	Data is collected for `methodA`; data for `methodX`, `methodY`, and `methodZ` is aggregated into an asterisk node. In the Call Graph, `methodA` calls an asterisk (*) node, which in turn calls `methodB`.	Using this policy, the calling method's data is preserved and you can see that the method made at least one call out to an encapsulated method. The names of the encapsulated methods are lost, but you are not interested in these methods anyway.
Full	For each encapsulated method called by a non-encapsulated method, JProbe aggregates the data for the encapsulated method plus any other encapsulated methods that it calls. In the Call Graph, these nodes contain a lock icon.	The data for `methodX`, `methodY`, and `methodZ` is aggregated and attributed to `methodX`. In the Call Graph, `methodA` calls `methodX`, which in turn calls `methodB`.	This policy provides the most detail. You can see the initial calls to encapsulated methods, the names of those methods, and data associated with each. With so much detail to collect, however, your analysis may take longer and the resulting Call Graph will be more complex.

Public Method Profiling

Data recorded during Performance analysis tracks method invocations included through filters and offers varying levels of detail for method calls that are not included by filters.

By adjusting the amount of detail on methods that are not included through filters, you can adjust the impact JProbe has on your application. The number of details you choose to record is directly proportional to the overhead your application may incur.

The Performance overhead can be greatly reduced by minimizing the number of tracked method invocations. Depending on the recording option you enable (at runtime or when

setting up the initial configuration), you select between tracking all method invocations included by their filters, or only public methods from their filters (*Public Methods, No Encapsulation*).

Triggers for Performance

You can select *Triggers* for your Performance analysis in the **Automation** tab of the configuration wizard.

Triggers provide a way to tie JProbe actions to your program's execution. You can tell JProbe to perform an action when your program enters or exits a method of your choice. With advanced trigger controls, you can also specify when to start executing the trigger actions and how long to execute them. Triggers are useful for automated sessions.

Triggers are defined in the same way for all the analysis tools, however the trigger actions are somewhat different. This section describes the trigger actions for a Performance analysis. To learn how to create triggers, see "Setting Triggers" on page 425.

Trigger Action	Description
Recording	Starts recording data. You can select an encapsulation option and the method profiling in the Recording Parameter area: • Data Recording Off • Public Methods, No Encapsulation • No Encapsulation • Grouped Encapsulation • Full Encapsulation **Note** When the trigger action is set to *Recording*, the selected recording parameter is also displayed in the triggers table.
Snapshot	Takes a snapshot. You can provide a base name for the snapshot; otherwise, the snapshot file name takes the form *snapshot_<n>_raw.jpp*.

12

Exploring Performance Analysis Views

This chapter provides detailed information about the Performance views used to present performance data.

This chapter contains the following sections:

Exploring the Runtime Summary View 274
Exploring the Summary View 283
Exploring the Methods View 297
Exploring the Method Detail View 325
Exploring the Performance Source View 335
Exploring the Performance Difference View 338
Setting Preferences for Performance Views 342

Exploring the Runtime Summary View

As you run a Performance session, the Runtime Summary view displays changes in counts in the Java heap. After you capture the data in a memory snapshot, you can use the heap investigation tools, starting with the Call Graph and Call Tree, to understand the performance issues in your program.

This section contains the following information:

- Understanding the Runtime Summary View
- Interacting with the Performance Runtime Summary View

Understanding the Runtime Summary View

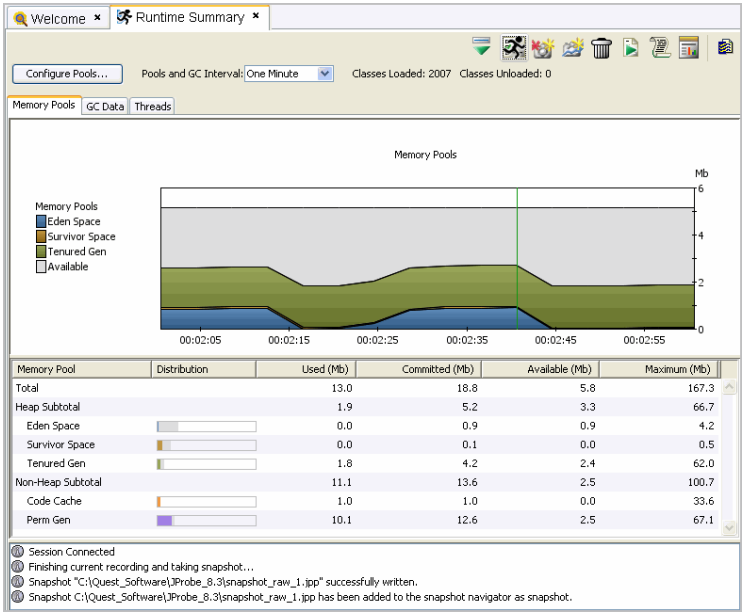
The Performance Runtime Summary view is both a view of the Java heap and a control center for running your analysis session. It helps you to identify deadlocks and also acts as a progress bar. A green vertical line on the Runtime Summary chart indicates a snapshot was taken, either due to a recording level change, or because a snapshot was requested.

After you have attached to a running session of your application, the Runtime Summary view appears. It contains the following tabs:

- Memory Pools Tab—displays memory usage in graphic and table formats.
- GC Data Tab—displays garbage collection data in graphic and table formats.
- Threads Tab—displays information about the application's threads.

Toolbar buttons provide access to the snapshot and garbage monitoring controls.

The system messages area (bottom pane) displays any errors and warnings that occur (including those that have occurred before this Console connected to the JProbe Analysis Engine).



Memory Pools Tab

The data displayed in the Memory Pools graph varies, depending on the JVM and the options that you are using. For JVMs that use generational garbage collectors, heap pools typically represent the generations. For example, in the Sun JVMs with some options, *Tenured Gen* is the old generation, while *Eden Space* and *Survivor Space* together make up the new generation.

While you are running an analysis session, the chart displays the size of the heap memory pools by default. Heap pools contain things normally found in the Java heap, like Java objects. Non-heap pools normally contain things the JVM uses internally, such as the compiled representation of method code or class information.

You can select a different time interval for the summary graph from the *Pools and GC Interval* list. The options are *One Minute*, *Five Minutes*, *Fifteen Minutes*, *Thirty Minutes*, and *One Hour.*

The Memory Pools table shows the actual or the estimated values for each of the heap pools in the graph. The column headings are defined as presented in the following table.

Used	The amount of memory currently being used.
Committed	The amount of memory that the OS has guaranteed to be available to the JVM. This is always greater than or equal to Used.
Available	The difference between Used and Committed memory.
Max	The maximum amount of memory that pool can use. Not all pools have a maximum value. Maximum value is often affected or directly set by JVM options like `-Xmx`. A Max value is always greater than the Committed value.

GC Data Tab

The **GC Data** tab contains a graph and a table. The graph displays each type of garbage collection at the time it occurred. The Count scale units at the right of the graph indicate how many collections were done of each type.

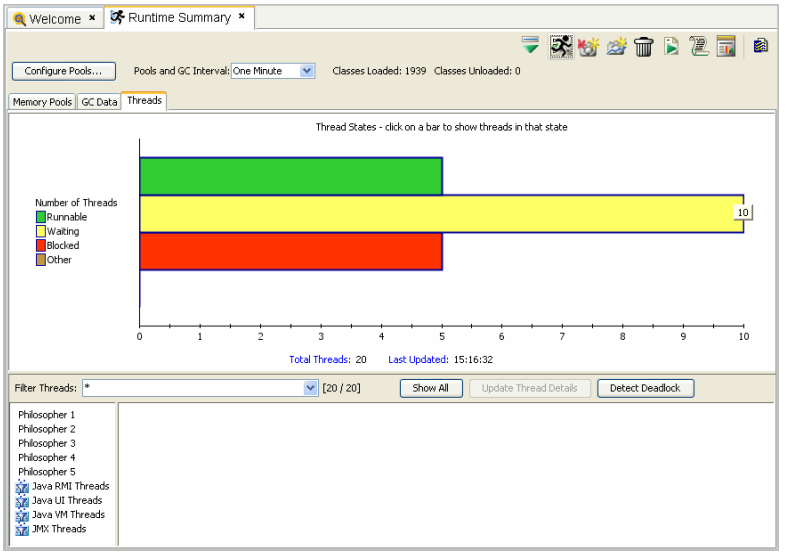
The *Garbage Collection Overhead* data below the graph indicates the percentage of time the JVM was exclusively performing garbage collection compared to the elapsed time the program has been running. For example, a program that has been running for 10 seconds and has spent 0.5 seconds doing garbage collection has a GC overhead of 5%. The lower the percentage, the lower the overhead. If the percentage is high, you may need to tune your JVM's GC options or reduce object cycling.

The table provides a breakdown by garbage collector that shows the number of times each collector has been invoked and the time (in milliseconds) that it used, as well as the cumulative counts and times.

Heap pools contain things normally found in the Java heap, like Java objects. Non-heap pools normally contain things the JVM uses internally, such as the compiled representation of method code or class information.

Threads Tab

The **Threads** tab contains a graph and a list, which are updated every four seconds.



The graph displays how many threads are in a given state at a certain time. The threads are grouped as presented in the following table.

Group	Thread State	Thread Description
Runnable	RUNNABLE	A thread executing in the Java virtual machine.
Waiting	TIMED_WAITING	A thread that is waiting for another thread to perform an action for up to a specified waiting time.
	WAITING	A thread that is waiting indefinitely for another thread to perform a particular action.
Blocked	BLOCKED	A thread that is blocked waiting for a monitor lock.
Other	NEW	A thread that has not started yet.
	TERMINATED	A thread that has exited.

Clicking on a bar or a line in the legend reduces the list of threads in the lower pane, so that only threads in that group are displayed. You can return to the full list by clicking the **Show All** button.

The total number of threads is displayed beneath the graph, along with the time the graph was last updated. Historical data is not maintained for the graph.

The lower pane displays a list of all known threads, grouped into categories. This list is sorted in ascending order (threads followed by categories). When you click a thread in the list, details for the selected thread are displayed on the right pane, including:

- Name of the thread.
- Thread state.
- The object upon which the thread is blocked, when applicable.
- The thread that owns the object that the thread is blocked, when applicable.
- Stack trace of the thread.
- List of object monitors locked by the thread.
- List of synchronizers locked by the thread.

You can reduce the list by clicking a bar in the graph, or by typing a filter into the **Filter Threads** field. The filter is a simple string filter that matches against thread names, not category names. Categories can be expanded by selecting them, right-clicking, and selecting **Expand Category**. Threads can be collapsed back into categories by selecting them, right-clicking, and selecting **Collapse Category**.

Thread details are only updated on demand. You can update the details for the currently selected thread by clicking the **Update Thread Details** button or by clicking a thread name. The button is disabled when no threads are selected or a category is selected.

You can detect deadlocks on demand, by clicking the **Detect Deadlock** button. If a deadlock is detected, the threads list displays the threads involved in the deadlock. If no deadlock is detected, a message is displayed to that effect.

Interacting with the Performance Runtime Summary View

The following table summarizes the tasks that you can perform.

Task	Action	Response
Change the data recording level or add a filter.	Click **Set Recording Level**  to open the *Set Recording Level* dialog box, then make the desired changes.	JProbe collects the selected level of data or applies the filter specified in the table. The current data set is saved to a snapshot. A new snapshot is started unless Data Recording was set to Off.
Take a snapshot	Click **Take Snapshot**  .	A snapshot of heap is taken. It appears in the Snapshots Navigator.
Take an HPROF heap dump	Click **Take an HPROF Heapdump**  .	An HPROF dump is taken, converted to a JProbe snapshot and added to the Snapshots Navigator. You can then analyze it using the JProbe Memory analysis tool.
Request a heap garbage collection	Click **Request Garbage Collection**  .	A garbage collection operation is performed and the results are displayed in the **GC Data** tab.
View the execution log from the Runtime Summary view	Click **Runtime Execution Log**  .	The **Execution Log** dialog box opens and you can review the information by navigating through the available tabs.

Task	Action	Response
View the history of events since the Console connected to the session.	Click **View Session History**  .	The **Session History** dialog box opens. In the left pane, you can parse the list and review the events that happened since the Console last connected to the session. In the right pane you can view additional details for select events.
Create a report for this view	Click **Generate Report**  to open the *Create Runtime Summary Report* dialog box. If you want to include in your report information about the threads currently displayed in the **Threads** tab, select the *Include Thread Information* check box. Select a location for saving the report and click **Generate File**.	The report is generated in the specified location.
Create a support bundle	Click **Create Support Bundle**  to automatically collect basic support information into a ZIP file, for submission to Quest Support.	The **Support Bundle Created** dialog box appears, indicating the location where the ZIP file was saved. You can click **Copy Bundle Name** to copy to the clipboard the name of the bundle that was just created.

Task	Action	Response
Display non-heap data	Click the **Configure Pools** button to open the *Select Memory Pools* dialog box, then select the **Non-Heap** option. **Note** This action may not be available for all JVMs.	The graphs refresh to display the non-heap data. Information on other data structures that are occupying memory (such as the permanent generation), may be included if the JVM supports it.
Change the data interval for the memory pools and garbage collections	Select the appropriate option from the *Pools and GC Interval* list.	The Memory Pools and Garbage Collection graphs change to the selected interval.
View thread details	In the **Threads** tab, click the **Update Thread Details** button or click a thread name. **Note** The button is disabled when no threads are selected or a category is selected.	JProbe displays the details for the currently selected thread.
Detect a deadlock	Click the **Detect Deadlock** button.	If a deadlock is detected, the threads list displays the threads involved in the deadlock. If no deadlock is detected, a message is displayed to that effect.
Save the chart as a PNG or JPEG file	Right-click in the chart and select **Save Chart As**, then select the graphic file type (**PNG** or **JPEG**) in which you want to save the chart. In the *Export Image* dialog box, specify the export parameters and a location to save the file.	The file is saved to the location you indicate.

Task	Action	Response
Export the chart to CSV	Right-click in the chart and select **Export to CSV**. In the *Export to CSV* dialog box, specify the export parameters and a location to save the file.	The file is saved to the location you indicate.
Show/hide table columns	Right-click the table and select **Show/Hide Columns**. In the *Show/Hide Columns* dialog box, select the columns to show and click **OK**.	The selected columns are displayed. The rest are hidden.
Resize column widths	To set the column width, drag the edge of the column heading left or right. To see all data in a column, right-click in the column and select **Size <column name> to Fit**. To fit all columns in the viewing area, right-click and select **Size All Columns to Fit**.	The columns are resized.
Rearrange columns	Drag a column by its heading and drop it in a new location.	The column appears in its new location.
Sort the table by a metric	Click the column heading for that metric. To reverse the sort order, click the heading again.	The table is sorted.
Export the table to CSV	Right-click in the table and select **Export to CSV**. Type a file name and click **Save**.	The file contains the table data separated by a comma. It may also contain headers. For more information, see "Changing How JProbe Exports to a CSV File" on page 455.

Exploring the Summary View

The Summary view presents an outline of the snapshot data and identifies potential performance hotspots. You can use this view for guidence on how to continue the performance analysis investigation.

This section contains the following information:

- "Opening the Summary View" on page 283
- "Understanding the Summary View" on page 285
- "Interacting with the Summary View" on page 292

Opening the Summary View

The Summary view is the first view you see when opening a performance snapshot.

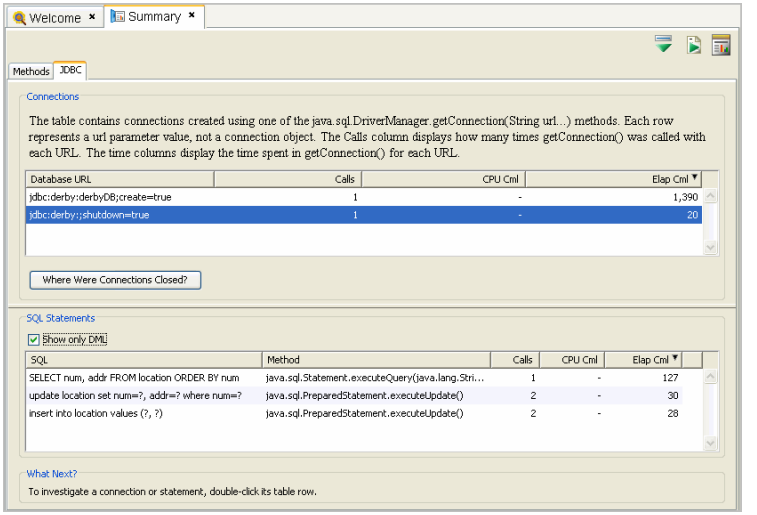
To open the Summary view:

- In the Snapshots Navigator, double-click the snapshot that you want to display.

  The Summary view opens.

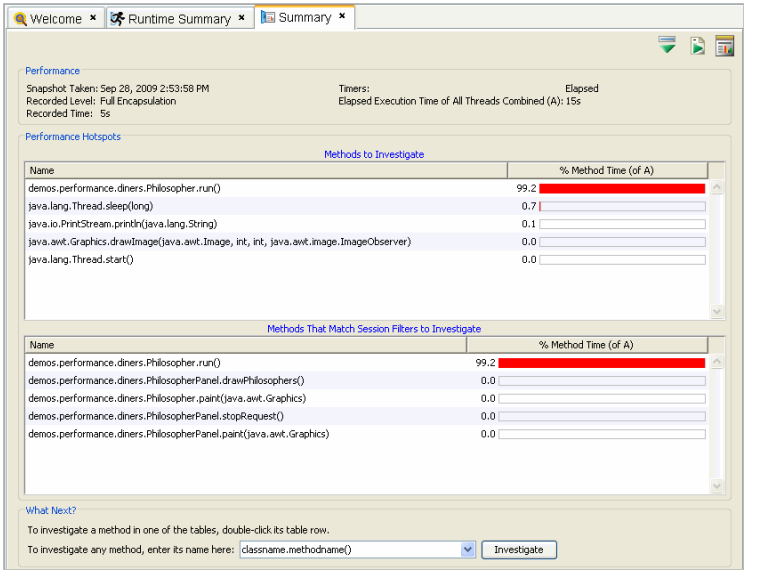
  If the snapshot contains JDBC data, the Summary view presents the snapshot's information in two tabs (**Methods** and **JDBC** tabs).

  The following illustration presents an example of a Summary view when JDBC data is recorded.

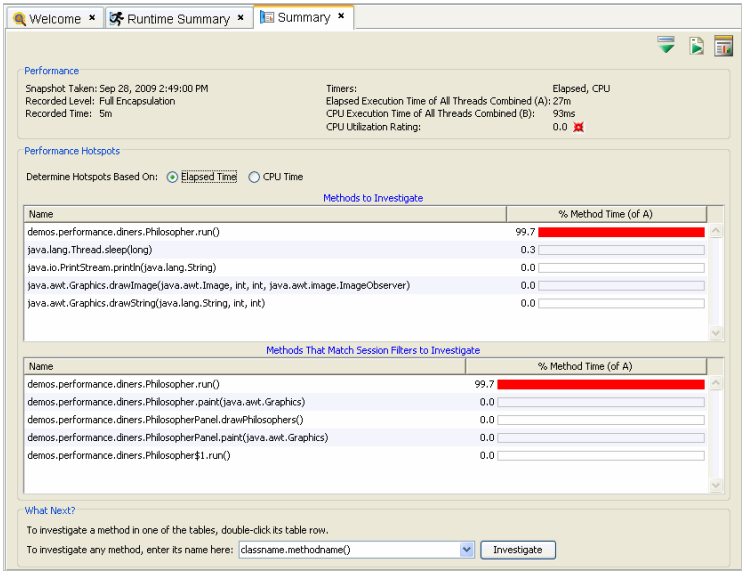


If the snapshot does not contain any JDBC data, the Summary view presents only the information relevant to Methods.

The following illustration presents an example of a Summary view when no JDBC data is recorded, and only the Elapsed Time data is available.



The following illustration presents an example of a Summary view when no JDBC data is recorded, and both Elapsed Time and CPU Time are available.



Understanding the Summary View

The Summary view contains the following tabs:

- Methods Tab—displays information that helps you identify potential performance hotspots and focus on problem methods.
- JDBC Tab—displays information related to JDBC connections and statement methods.

  Important This tab is displayed only if JDBC data is recorded in the snapshot.

Toolbar buttons provide access to the execution log and to the report creation dialog.

Methods Tab

The **Methods** tab is pre-populated with information that helps you identify potential performance hotspots and focus on problem methods. From this tab, you can navigate to subsequent views, which provide detailed information about the selected methods of interest.

The *Application* section (upper part of the view) presents global performance data about the session that was run to obtain the snapshot and about the snapshot itself. The following table lists the type of information displayed in the *Application* section.

Parameter	Information Provided	Notes
<ApplicationName>	Name of the application being analyzed.	N/A
Snapshot Taken	Date and time when the snapshot was taken.	N/A
Recorded Level	Performance recording level used when capturing the snapshot.	N/A
Recorded Time	Actual recording time (clock time).	N/A
Timers	Timers available (Elapsed or Elapsed/CPU).	N/A
Elapsed Execution Time of All Threads Combined (A)	Sum of “all threads’ elapsed cumulative times” minus the sum of “all threads’ elapsed method times”.	N/A
CPU Execution Time of All Threads Combined (B)	Sum of “all threads’ CPU cumulative times” minus the sum of “all threads’ CPU method times”.	Available only when collecting CPU data.
CPU Utilization Rating	CPU utilization rating value and icon (**Good** , **OK** , **Bad** , or **Severe**).	Available only when collecting CPU data.

The CPU Utilization Rating is defined as the *CPU Execution Time of all Threads Combined* divided by the *Recorded Time*. Four CPU Utilization Rating ranges are defined, as follows:

Range	Rating
0.5 < CPU Utilization Rating	Good
0.25 < CPU UR <= 0.5	OK
0.1 < CPU UR <= 0.25	Bad
CPU UR <= 0.1	Severe

The *Performance Hotspots* section suggests methods to investigate based on Elapsed Time data or Elapsed Time and CPU Time data (when available). In the latter case, you can switch between the two types of data by selecting the **CPU Time** or **Elapsed Time** option button, respectively.

Note The **CPU Time** and **Elapsed Time** option buttons are displayed only when CPU data is available. The CPU data is collected if the CPU timing option is selected when defining the application settings (for details, see "Timing Settings for Performance" on page 266).

The hotspots are determined by searching for the top five most expensive methods, by method time. The upper hotspots table contains the top five methods from all of the methods for which we have data available.

Tip If the recording level is set to `grouped`, grouped methods are omitted from this table, as they are not a good starting point for investigation.

The lower hotspots table contains the top five methods that match the session filters defined by users when collecting data. Therefore, this table contains only non-encapsulated and non-grouped methods that match the filters set in the Create/Edit settings dialog, on the **Analysis Type > Data Collection Filters** tab.

The following table lists the type of information displayed in the *Performance Hotspots* section.

Parameter	Information Provided	Notes
Methods To Investigate	This table contains five suggested methods to investigate based on elapsed time data.	N/A
	This table contains five suggested methods to investigate based on CPU time data.	Available only when determining hotspots based on CPU time.
Methods That Match Session Filters to Investigate	This table contains five suggested methods, that match the session filters, to investigate based on elapsed time data.	N/A
	This table contains five suggested methods, that match the session filters, to investigate based on CPU time data.	Available only when determining hotspots based on CPU time.

Each row in these tables contains a method *Name* and the *% Method Time*, which is defined as follows:

- When the Elapsed Time option is selected:

  ```
  (method elapsed time/elapsed execution time of all threads
  combined)*100
  ```

- When CPU Time option is selected:

  ```
  (method cpu time/cpu execution time of all threads
  combined)*100
  ```

Cells that contain percentage values also contain percentage bars. The tables are not sortable, and they always display only five rows.

Double-clicking a row opens the **Methods** view, with the method in that row as the focus method.

Important If you select a new method to investigate when the **Methods** view is already open, the **Methods** view is reloaded with the new method as the focus method. All history on the **Methods** view is lost.

The *What Next?* section guides you on how to continue the performance analysis investigation. You can select the method to investigate in the Performance Hotspots tables and click **Investigate** to open the **Methods** view with the method in that row as the focus method.

If you want to continue the analysis with a method that is not suggested in the Performance Hotspots tables, you can type a filter into the text box provided in this section, and click **Investigate**. In the case that multiple methods match this filter, the **Matching Methods** dialog box opens, displaying all the methods that match the filter. From here, you can select a method and open the **Methods** view.

Note The Console remembers the last five filters and displays them in the **Investigate** list.

JDBC Tab

The **JDBC** tab is populated with information related to JDBC connections and statements. From this tab, you can navigate to subsequent views, which provide detailed information about methods that offered the connection or statement information.

Important This tab is displayed only if JDBC data is recorded in the snapshot.

The *Connections* section presents the connections created using one of the `java.sql.Driver` or `java.sql.DriverManager getConnection(...)`

methods. The following table lists the type of information displayed in the *Connections* table.

Parameter	Information Provided
Database URL	Represents the URL connection string (or, if no URL was provided, the class of the driver that was instantiated), truncated to 50 characters. JProbe provides several truncation options. For details, see Setting Preferences for Performance Views. **Note** This is the URL parameter value, not the connection object.
Calls	Displays how many times `getConnection()` was called with each URL.
CPU Cml	Displays the CPU time spent in `getConnection()` for each URL.
Elap Cml	Displays the elapsed time spent in `getConnection()` for each URL.

By double-clicking a row in the *Connections* section you open the Methods view with the graph associated with the row as the focus node. By clicking the **Where Were Connections Closed?** button you open the Methods view with the `Connection.close()` method as the focus node.

The *SQL Statements* section displays the time required to execute SQL statements. The following table lists the type of information displayed in the *SQL Statements* table.

Parameter	Information Provided
SQL	Represents the SQL statement, truncated to 50 characters. JProbe provides several truncation options. For details, see Setting Preferences for Performance Views. **Note** Each instance of a prepared or a callable statement that uses a particular SQL string is represented by its own row. However, batch execution (which contains more than one SQL statement) is represented in the SQL column only by an asterisk (*). Select the row, right-click, and select **View SQL Statements** to display all the statements represented by that row.
Method	Displays the name of the JDBC method used to execute the query.
Calls	Displays how many times the JDBC method was called.
CPU Cml	Displays the CPU cumulative time.
Elap Cml	Displays the elapsed cumulative time.

By double-clicking a row in the *SQL Statements* section you open the Methods view with the graph associated with the row as the focus node. By selecting a row in the *SQL Statements* section, right-clicking, and selecting the appropriate option, you can investigate where the selected statements were executed or the statement's SQL was collected.

The *Show only DML* check box is by default selected, restricting the SQL statements shown to the appropriate statement types (that is, only a subset of SQL statements—known as DML (Data Manipulation Language)—is displayed). To display all collected SQL statements, clear the *Show only DML* check box. DML statement types are appropriate for tuning using Quest's SQL Optimizer for Oracle.

To tune an SQL statement:

1. Install Quest's SQL Optimizer for Oracle on the machine that will run the JProbe Console.

   Note For information about this product, refer to the SQL Optimizer for Oracle documentation set.

2 Open the performance snapshot containing SQL statements in the JProbe Console.

3 In the **Summary** view > **JDBC** tab, right-click the statement that needs tuning and select *Tune SQL Statement.*

The SQL Optimizer for Oracle opens with the SQL text already inserted.

Note You can repeat this step for all SQL statements in the list, with the exception of batch queries.

4 Run the SQL Optimizer for Oracle to tune your SQL statement and identify the "optimal" one.

You can update your SQL statement(s) in the original source by using the tuned statements, then run your application again with JProbe.

The *What Next?* section guides you to choose a connection or statement to investigate. The graph node associated with the connection or statement is the focus node for the Methods view that opens when you continue your Performance analysis.

Interacting with the Summary View

Several operations are available from the Summary view's tabs, from the toolbar buttons, or the toolbar menu. For details, see the following topics:

- Exploring the Summary View Toolbar
- Exploring the Methods Tab
- Exploring the JDBC Tab

Exploring the Summary View Toolbar

The following table lists the operations that you can perform from the Summary view toolbar.

Task	Action	Response
View the execution log from the Summary view	Click **Execution Log** .	The **Execution Log** dialog box opens and you can review the information by navigating through the available tabs.

Task	Action	Response
Create a report for this view	Click **Generate Report**  to open the *Create Summary Report* dialog box. Select a location for saving the report and click **Generate File**.	The report is generated in the specified location.

Exploring the Methods Tab

The following table lists the operations that you can perform in the **Methods** tab.

Task	Action	Response
Move to the **Methods** view with a focus method	Double-click a row in the *Performance Hotspots* tables. OR Select a row in the *Performance Hotspots* table, right-click, and select **Investigate**. OR In the *What Next?* section, type in a method name in the field, or select one from the list, and click **Investigate**.	The Methods view opens with the selected method as the focus method.
Resize column widths	To set the column width, drag the edge of the column heading left or right. To see all data in a column, right-click in the column and select **Size <column name> to Fit**. To fit all columns in the viewing area, right-click and select **Size All Columns to Fit**.	The columns are resized.

Task	Action	Response
Export the table to CSV	Right-click in the table and select **Export to CSV**. Type a file name and click **Save**.	The file contains the table data separated by a comma. It may also contain headers. For more information, see "Changing How JProbe Exports to a CSV File" on page 455.

Exploring the JDBC Tab

Important This tab is displayed only if JDBC data is recorded in the snapshot.

The following table lists the operations that you can perform in the **JDBC** tab.

Task	Action	Response
Investigate a connection	Double-click a row in the *Connections* table. OR Select a row in the *Connections* table, right-click, and select **Investigate**.	The Methods view opens with the selected graph node associated with the row as the focus method.
Investigate closed connections	Select a row in the *Connections* table and click the **Where Were Connections Closed?** button.	The Methods view opens with the *Connection.close()* method as the focus node.
Investigate a statement	Double-click a row in the *SQL Statements* table.	The Methods view opens with the selected graph node associated with the row as the focus method.

Task	Action	Response
Investigate where a statement was executed	Select a row in the *SQL Statements* table, right-click, and select **Where was this Executed?**.	The Methods view opens with the *java.sql.Statement.execute()* method as the focus node.
Investigate where a statement was collected	Select a row in the *SQL Statements* table, right-click, and select **Where was this Collected?**.	The Methods view opens with the *java.sql.Statement.execute()* method as the focus node.
Tune an SQL statement	Select a row in the *SQL Statements* table, right-click, and select **Tune SQL Statement**. **Note** This option is available only if you have already installed Quest's SQL Optimizer for Oracle on the machine that will run the JProbe Console.	The SQL Optimizer for Oracle opens with the SQL text already inserted. You can run the tool to optimize the SQL statement.
View grouped SQL statements **Note** When a row in the table represents more than one SQL statement, the SQL column displays only an asterisk (*).	Select the asterisk (*) row in the *SQL Statements* table, right-click, and select **View SQL Statements**.	The SQL Statements dialog box opens, displaying all the statements represented by the selected row.
Show/hide table columns	Right-click in the column and select **Show/Hide Columns**. Select the columns to show and click **OK**.	The selected columns are displayed. The rest are hidden.

Task	Action	Response
Hide the column currently selected	Right-click in the column and select **Hide Column <column name>**.	The column currently selected (that is, the column based on which the table is sorted) is hidden.
Resize column widths	To set the column width, drag the edge of the column heading left or right. To see all data in a column, right-click in the column and select **Size <column name> to Fit**. To fit all columns in the viewing area, right-click and select **Size All Columns to Fit**.	The columns are resized.
Export the table to CSV	Right-click in the table and select **Export to CSV**. Type a file name and click **Save**.	The file contains the table data separated by a comma. It may also contain headers. For more information, see “Changing How JProbe Exports to a CSV File” on page 455.

Exploring the Methods View

The Methods view is a powerful tool for viewing detailed information about the focused method selected in the Summary view and its calling relationship with other methods in your application. Its features are discussed in the following sections:

- Opening the Methods View
- Understanding the Methods View
- Interacting with the Methods View
- Customizing the Call Graph

Opening the Methods View

The Methods view can be reached from the Performance Summary view.

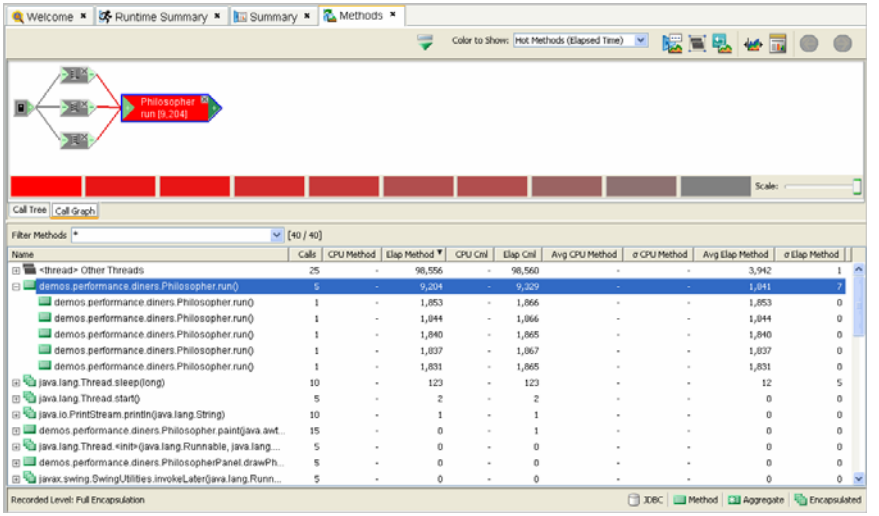
To open the Methods view:

- In the Summary view, double-click the method you want display.

  The Methods view opens with the selected method as the focus method. The view has two ways to visualize the calling relationships: a call graph and a call tree, as well as a summary view of the data in a method list, which contains a list of your methods and threads (if included) with data for all the metrics.

Understanding the Methods View

The Methods view consists of two sections: the upper pane contains the Call Graph and Call Tree tabs. The lower pane contains the Method List.



Call Graph

The Call Graph starts from the node provided to this view by the Performance Summary view. The Call Graph is a directed graph that shows the calling relationships among executed methods in your program. The nodes represent the methods, and the edges represent the call from one method to another. Some nodes are conceptual in nature; in particular, the root of the graph and the thread nodes. Other nodes are method types, including standard methods and initialization methods. By default, thread nodes are hidden and are not included in calculations.

It can be helpful to think of the Call Graph as a set of superimposed call trees. For example, if two methods call methodA, then methodA appears as a single node with two parents. If you remove the call tree for one parent (this is called pruning), methodA remains because it has another parent.

The following table lists the types of nodes found in the Call Graph.

Icon	Type	Description
	Root	Conceptual in nature, the root node is the ancestor of all threads (or call trees if the thread nodes are hidden).

Icon	Type	Description
	Thread	Conceptual in nature, a thread node is the ancestor of the methods invoked by that thread.
	Encapsulated	Represents methods that you are not interested in analyzing, based on the encapsulated data collection filter.
	<Class>.<method>	Represents a method for which you collected data at either method-level or line-level.
as above	<Class>.`<init>`()	`javac` compiles your constructors into initialization methods.
as above	<Class>.`<clinit>`()	`javac` compiles your class initialization statements into <clinit>() methods that are executed when the class is initialized by the JVM.
	Simple	Represents small Java methods, typically accessors (get/set), that are generally too simple for timed analysis. JProbe automatically recognizes these methods. Methods which include operations such as looping, object creation, exception handling, or calls to other methods (including constructors) are not considered simple. The time for simple methods displays as 0.

Setting filters helps to narrow your investigation. When you set filters and run a session, you will see nodes in the call graph that represent the types of filters that you set.

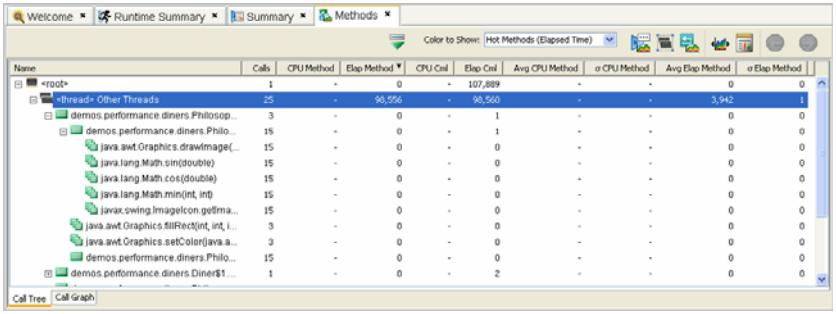
Icon	Type	Description
	Line	This type of node (includes a gray box with lines) indicates line level profiling information. It displays line by line how the method time is spent. When you place your cursor over the icon the tooltip reveals the information for the methods. Method type, cumulative time, and, number of calls. Typically, using this type of filter is expensive.
	Method	This type of node indicates the method which is being filtered and any parameters associated with this method.

Icon	Type	Description
	None	This type of node indicates that the method may have called other methods that are not shown in the graph. This happens for native methods and for methods collected while the encapsulation level is set to *None*. In this case, you can view the list of methods represented by this node, by selecting the node, right-clicking, and selecting **View Method Names**. For normal method nodes captured without encapsulation, the View Method Names list represents encapsulated methods that the selected method may have called. Those encapsulated methods may have also called other encapsulated methods. The Method Time calculated in this case for the normal method node includes the named method, and possibly time from none, some, or all of the listed methods, and other encapsulated methods they called. For grouped nodes, the View Method Names list represents encapsulated methods that the parent method may have called. The Method Time calculated in this case for the grouped node includes time from some or all of the listed methods, and any encapsulated methods they called.

Icon	Type	Description
 or  or 	Encapsulated - Grouped	These types of nodes indicate that a method is encapsulated. The **lock** represents an encapsulated method. The **lightning bolt** represents a simple method. The **asterisk** represents a method that is encapsulated using the group type. Details about calls to other methods or from other methods are not shown, but their time and allocated objects are held by this node. This node also includes native methods; those that are implemented in some other language than Java. For more information, see How Data is Affected by Using Performance Features. In this case, you can select a method, right-click, and select **View Method Names**, to show the methods that the parent may have called and that may be included in this node.

Call Tree

The Call Tree presents the same data as the Call Graph in a traditional, more detailed tree view. While the graph displays only one node per method, the tree view has a node for every time a method is called and displays the nodes in the order of the calls.



Each row in the Call Tree represents a method called by your program during its execution. Searching is not available in the tree, but you can sort the columns to find the methods that are using the largest amounts of time.

A time value of zero (0) can be caused by three things:

- Rounding—occurs when the timing unit or precision does not allow for enough decimal places to display the precise value. Very small amounts of time can be rounded down to zero. Rounding is a display issue only; calculations are not affected. Changing the Time Unit may prevent rounding (see “Customizing the Call Graph” on page 323).
- Insufficient timing resolution—means that the fraction of time spent in the method is less than the time measurable by JProbe.
- A method that uses a small amount of time.

The name of each method in the Call Tree is prefixed by an icon which provides details about that method, as described in the following table.

Icon	Type
	Root nodes (optional).
	Thread nodes.
	Grouped thread nodes.
	Normal/ simple nodes.
	Encapsulated/ grouped nodes. **Note** The methods' names match their parent's names.
	Aggregate nodes.
	JDBC nodes.

Note The status bar also displays the most common icons and short descriptions of what they represent.

When you use the prune or isolate functions in the Call Graph, the corresponding nodes are removed from the Call Tree. However, if you use those functions in the Call Tree, the Call Graph is not changed, although the underlying data for the corresponding nodes does change. The only exception is if the corresponding node in the graph represents only one call; in that case, it is removed from the graph.

Method List

The Method List is defined as a fixed two-level tree table. The top level represents a graph node and the second level represents a CCT node. A selection in the Method List can be mapped to the Call Graph and Call Tree, and vice versa.

Note When the Methods view is first opened, the Call Graph displays five nodes. To display additional nodes, use the **Show More Nodes** toolbar button (see section "Exploring the Methods View Toolbar" on page 306).

Each row in the list represents a method called by your program during its execution. The data set may have fewer methods if you edited the data set by pruning subtrees or by isolating a subtree. The Filter Methods field enables you to filter data based on the package or class of a selected method. If a method in the list is not included in the Call Graph, you can add it.

The name of each row in the Methods List is prefixed by an icon which indicates information about the method, as described in the Call Tree section.

Interacting with the Methods View

Several operations are available from the context menu of a selected item in any of the three components of the Methods view (that is, Call Graph, Call Tree, and Methods List), from the toolbar buttons, or the toolbar menu. For details, see the following topics:

- "Exploring the Methods View Toolbar" on page 306
- "Exploring the Call Graph" on page 308
- "Exploring the Call Tree" on page 312
- "Exploring the Method List" on page 315
- "Editing the Data Set" on page 318
- "Navigating the Graph by Showing and Hiding Nodes" on page 320

The interactions with and among the Call Graph, Call Tree, and Method List are designed in such a way that you can quickly locate nodes in other sections of the view. For example, you can:

- Switch between the Call Graph and the Call Tree while maintaining context.
- Find methods in the Methods List, by selecting a node in the Call Graph, which also selects the same node in the Methods List.

- Find methods in the Call Tree, by selecting a CCT node in the Method List, which quickly finds the same node in the Call Tree.

Exploring the Methods View Toolbar

The following table lists the operations that you can perform from the Methods view toolbar.

Task	Action	Response
Show more graph nodes	Click **Show More Nodes**  .	The Call Graph is updated to display more nodes along the critical path, until you reach the root. **Note** To return to the default value (that is, top five nodes displayed), use the Back history operation.
Group/ungroup thread nodes	Click **Do Not Group Threads**  .	Displays threads grouped or ungrouped in the graph and trees.
Restore the graph to the “before edits” state	Click **Reset**  .	Resets the graph to its default display criteria and clears history of operations. All methods removed through pruning or isolating are restored, top methods that were hidden are shown, and all other nodes that were shown are hidden. You cannot undo a reset action.

Task	Action	Response
Move around in a large graph	Click **Show Graph Panner**  to open the Graph Panner window, then move the rectangle to the part of the graph that you want to see.	The Call Graph shows the section of the graph that you selected.
Create a report for this view	Click **Generate Report**  to open the *Create Methods Report* dialog box. Select a location for saving the report and click **Generate File**.	The report is generated in the specified location.
Undo an edit	Click **Back**  .	The Method view reverts to the state before the edit took place. You can undo reset, isolate, and prune actions. **Note** You can not undo the reset action.
Redo the last undo	Click **Forward**  .	Reapplies the edit on the view.
Customize the fonts	See “Customizing Fonts” on page 324.	Changes the font family, size, and style of text in the graph.

Task	Action	Response
Set the metric by which nodes are colored	Select one of the options available in the *Color to Show* list, as necessary: • Hot Paths (Elapsed Time/CPU Time)—to display the call tree that has the largest impact overall on program execution time (critical path). • Hot Methods (Elapsed Time/ CPU Time)—to display individual methods that have the largest impact overall on program execution time. • Hot Methods (Number of Calls)—to find excessive method calls. **Note** The *Color to Show* list is by default initialized to Hot Methods (Elapsed Time/ CPU Time).	The currently displayed set of nodes are colored according to the metric you selected, and the color scale tooltips now reflect values for this metric. In addition, node tooltips reflect values for the selected metric. In the Method List, the heading that corresponds to the selected metric is highlighted in a bright color. If the column was previously hidden, it is automatically displayed and highlighted.

Exploring the Call Graph

You can explore a graph by scrolling or using the graph panner. You can see more or less of a graph by zooming in or out on the graph or by changing the physical size of the graph pane.

The following table describes how you can explore the Call Graph.

Task	Action	Response
Increase the area of the graph	Drag the bottom edge down.	The graph resizes to fit the larger area.

Task	Action	Response
Move around in a large graph	Do one of the following: • Scroll using the scroll bars across the bottom and to the right of the graph. • Click **Show Graph Panner**  to open the Graph Panner window, then move the rectangle to the part of the graph that you want to see.	The Call Graph shows the section of the graph that you selected.
Zoom out to see more of the graph	Move the *Scale* slider to the left.	The scale of the graph is reduced to display more nodes.
Zoom in to focus on a few nodes	Move the *Scale* slider to the right.	The scale of the graph is increased and fewer nodes are shown.
Clean up the display of the nodes in the graph	Right-click the graph and select **Layout Graph**.	The nodes are rearranged to improve readability.
Save graph as a PNG or JPEG file	Right-click in the graph and select **Save Graph As**, then select the graphic file type (**PNG** or **JPEG**) in which you want to save the graph. In the *Export Image* dialog box, specify the export parameters and a location to save the file.	The file is saved to the location you indicate.

The following table lists the operations that you can perform from the context menu of a Call Graph node.

Task	Action: select a node in the Call Graph tab, right-click, and select...	Response
Prune a subtree	**Prune Selected Node**.	The selected node and its subtree are removed from the graph. All metrics are recalculated without the contribution of the pruned call tree. For more information, see “Pruning Subtrees” on page 319.
Isolate a subtree	**Isolate Selected Node**.	The selected node is anchored to the root node. All nodes that do not belong to the method’s subtree are removed. All metrics are recalculated to include only the data collected on the methods in the isolated tree. For more information, see “Isolating Subtrees” on page 319.
Re-draw the graph using the select node as the focus method	**Redraw Graph Based On Selected Node**.	Resets the graph (clears the history) and re-draws it using this node as the focus method.
Move to the Method Detail view with a focus method	**Open Method Detail View**.	The Method Detail view opens with the selected method as the focus method.

Task	Action: select a node in the Call Graph tab, right-click, and select...	Response
View the list of methods represented by this node	**View Method Names**.	Opens the Methods dialog box, which contains a list of method names represented by this grouped node.
Drill down to source code	**Show Source**.	If you have the source code, the Source view opens with the source code displayed.
Drill down to line data	**Show Line Data**.	The Source view associated with the selected method opens. Disabled if line data is not available for the selected method.

The following table lists the operations that you can perform from the context menu of a Call Graph edge.

Task	Action: select an edge in the Call Graph tab, right-click, and select...	Response
Find the destination node	**Go to Destination Node**.	Selects the destination node (that is, the node to which the arrow points). **Note** The node is highlighted in all three components (Call Graph, Call Tree, and Method List).

Task	Action: select an edge in the Call Graph tab, right-click, and select...	Response
Find the source node	**Go to Source Node**.	Selects the source node (that is, the node from which the arrow originates). **Note** The node is highlighted in all three components (Call Graph, Call Tree, and Method List).

Exploring the Call Tree

The following table lists the operations that you can perform from the Call Tree context menu.

Task	Action: select a node in the Call Tree tab, right-click, and select...	Response
Expand a node to the next branch point	**Expand To Next Branch Point**.	The selected node expands to the next branch point.
Move to the Method Detail view with a focus method	**Open Method Detail View**.	The Method Detail view opens with the selected method as the focus method.
View the list of threads represented by this node	**View Thread Names**.	Opens the Threads dialog box, which contains a list of thread names represented by this grouped node.

Task	Action: select a node in the Call Tree tab, right-click, and select...	Response
View the list of methods represented by this node	**View Method Names**.	Opens the Methods dialog box, which contains a list of method names represented by this grouped node.
Drill down to source code	**Show Source**.	If you have the source code, the Source view opens with the source code displayed.
Drill down to line data	**Show Line Data**.	The Source view associated with the selected method opens. Disabled if line data is not available for the selected method.
Prune a subtree	**Prune Selected Node**.	The selected node and its subtree are removed from the graph. All metrics are recalculated without the contribution of the pruned call tree. For more information, see “Pruning Subtrees” on page 319.

Task	Action: select a node in the Call Tree tab, right-click, and select...	Response
Isolate a subtree	**Isolate Selected Node**.	The selected node is anchored to the root node. All nodes that do not belong to the method's subtree are removed. All metrics are recalculated to include only the data collected on the methods in the isolated tree. For more information, see "Isolating Subtrees" on page 319.
Show/hide table columns	**Show/Hide Columns**. Select the columns to show and click **OK**.	The selected columns are displayed. The rest are hidden.
Show/hide all columns displaying Elapsed Times	**Show All Elapsed Columns**.	All columns displaying Elapsed Times are displayed/hidden.
Show/hide all columns displaying CPU Times	**Show All CPU Columns**.	All columns displaying CPU Times are displayed/ hidden, if this information is available.
Hide the column currently selected	**Hide Column <column name>**.	The column currently selected (that is, the column based on which the Call Tree table is sorted) is hidden.

Task	Action: select a node in the Call Tree tab, right-click, and select...	Response
Resize column widths	To set the width, drag the edge of the column heading left or right. To see all data in a column, right-click in the column and select **Size <column name> to Fit**. To fit all columns in the viewing area, right-click and select **Size All Columns to Fit**.	The columns are resized.
Sort the table by a metric	Click the column heading for the metric.	The table is sorted. To reverse the sort order, click the heading again.

Exploring the Method List

The following table lists the operations that you can perform from the Method List context menu.

Task	Action: select a node in the Method List, right-click, and select...	Response
Add a method to the Call Graph	**Fit Method Into Graph**.	The method is added as a node to the Call Graph. Any other previously hidden nodes that are required to link the new node to the existing graph are also added.
Move to the Method Detail view with a focus method	**Open Method Detail View**.	The Method Detail view opens with the selected method as the focus method.

Task	Action: select a node in the Method List, right-click, and select...	Response
View the list of threads represented by this node	**View Thread Names**.	Opens the Threads dialog box, which contains a list of thread names represented by this grouped node.
View the list of methods represented by this node	**View Method Names**.	Opens the Methods dialog box, which contains a list of method names represented by this grouped node.
Drill down to source code	**Show Source**.	If you have the source code, the Source view opens with the source code displayed.
Drill down to line data	**Show Line Data**.	The Source view associated with the selected method opens. Disabled if line data is not available for the selected method.
Set filter based on the method's package or class	**Set Filter Based On** • Method Package • Method Class	The Method List is updated to display only the methods that pass the set filter. The *Filter Methods* field above the Method List displays the name of the selected method package or class.

Task	Action: select a node in the Method List, right-click, and select...	Response
Prune a subtree	**Prune Selected Node**.	The selected node and its subtree are removed from the graph. All metrics are recalculated without the contribution of the pruned call tree. For more information, see “Pruning Subtrees” on page 319.
Isolate a subtree	**Isolate Selected Node**.	The selected node is anchored to the root node. All nodes that do not belong to the method’s subtree are removed. All metrics are recalculated to include only the data collected on the methods in the isolated tree. For more information, see “Isolating Subtrees” on page 319.
Re-draw the graph using the select node as the focus method	**Redraw Graph Based On Selected Node**.	Resets the graph (clears the history) and re-draws it using this node as the focus method.
Show/hide table columns	**Show/Hide Columns**. Select the columns to show and click **OK**.	The selected columns are displayed. The rest are hidden.
Show/hide all columns displaying Elapsed Times	**Show All Elapsed Columns**.	All columns displaying Elapsed Times are displayed/hidden.

Task	Action: select a node in the Method List, right-click, and select...	Response
Show/hide all columns displaying CPU Times	**Show All CPU Columns**.	All columns displaying CPU Times are displayed/ hidden, if this information is available.
Hide the column currently selected	**Hide Column <column name>**.	The column currently selected (that is, the column based on which the Call Tree table is sorted) is hidden.
Resize column widths	To set the width, drag the edge of the column heading left or right. To see all data in a column, right-click in the column and select **Size <column name> to Fit**. To fit all columns in the viewing area, right-click and select **Size All Columns to Fit**.	The columns are resized.
Sort the table by a metric	Click the column heading for the metric.	The table is sorted. To reverse the sort order, click the heading again.

Editing the Data Set

The data set in the snapshot reflects the data collected during the session. You can temporarily remove data from the current data set by pruning some trees or isolating a single tree. All the calculated metrics are automatically recalculated for the data that remains. JProbe keeps a history of all edits to the data set that is separate from the history it keeps for hide/show actions. You can revert to the entire data set by resetting the graph.

For more information, see the following topics:

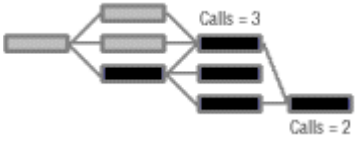
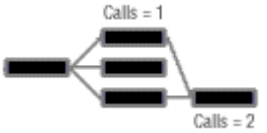
- Isolating Subtrees

- Pruning Subtrees

Isolating Subtrees

You can manually edit the data set by isolating a call tree. For example, if you want to focus on a single thread or the subtree of a method, you isolate the tree in the Call Graph. The method is anchored to the root node and the data set is recalculated to represent only that tree. For examples of when you might use isolate, see “Focusing on Threads and Methods in the Call Graph and Call Tree” on page 249.

The following diagrams show a Call Graph before and after a subtree is isolated. The dark nodes represent the target subtree.

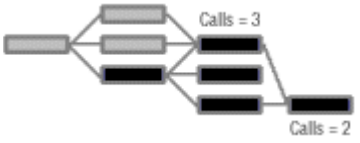
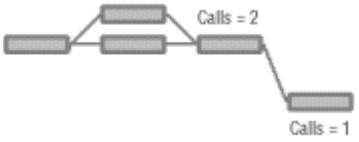
Original Call Graph	After Isolating Subtree
	

Note Line-level data is not recalculated. While the Call Graph displays the newly recalculated method-level data, the Source view continues to show the original line data.

Pruning Subtrees

You can selectively edit the data set by removing call trees from the Call Graph. Essentially, the graph is a set of superimposed subtrees, each representing a series of calls. You can remove layers of subtrees by pruning them from the graph. The data set is recalculated to eliminate the contribution of the removed methods. For examples of when you might use prune, see “Removing Threads and Method Subtrees” on page 250.

In the following example, the dark nodes represent the subtree to be pruned away. Notice that methods that were also called by other trees remain in the graph, though the number of calls attributed to the nodes decreases by the number of calls associated with the pruned tree; in this case, by one each.

Original Call Graph	After Pruning Subtree
	

Note Line-level data is not recalculated. While the Call Graph displays the newly recalculated method-level data, the Source view continues to show the original line data.

Navigating the Graph by Showing and Hiding Nodes

You can change the view of the graph without editing the data set. JProbe keeps a history of changes to the view and also a history of edits to the data set.

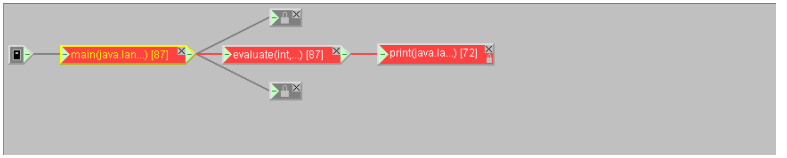
For more information, see the following topics:

- "Hiding and Showing Nodes" on page 320
- "Navigating Subgraphs" on page 321
- "Navigating Calling Methods" on page 322
- "Going Back to Previous Views of the Graph" on page 323

Hiding and Showing Nodes

You can hide and show a node, and you can show a single node with only its top parent and child methods.

Note Hiding and showing nodes does not change the data set. To remove some data from the data set, use isolate or prune. For details, see "Editing the Data Set" on page 318.



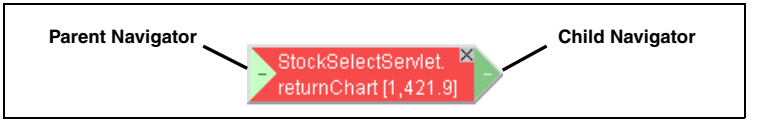
The following table lists the node-related tasks that you can perform.

Task	Action	Response
Hide a method (node)	Click the **X** on the node.	The node disappears. If the node had children that were not called by any other method, the children are hidden.
Show a hidden method	Do one of the following: • If you just hid the node, click **Back** to go back to the previous graph. • Click the + on the parent node.	The node is displayed, as well as any other previously hidden nodes that are required to link the new node to the existing graph.
Clean up the display of the nodes in the graph	Right-click the graph and select **Layout Graph**.	The nodes are rearranged to improve readability.

Navigating Subgraphs

While investigating hotspots, you usually need to show more than the top methods. For example, you may want to show the top three methods called by a hotspot method to see which child method contributes the most to the method's cumulative time. In other cases, you may need to show an entire subgraph to find the real cause of a hotspot. You can also hide subgraphs after you are sure the methods are not contributing to your performance bottleneck. Hiding subgraphs reduces the complexity of the graph, making it easier for you to follow the path you are interested in analyzing.

Use the child navigator to hide and show the methods that are called by a method. The navigators are illustrated in the diagram below.



The following table lists the subgraph-related tasks that you can perform.

Child Navigator Task	Action	Response
Hide a subgraph	Click the child navigator (default action if all top children are shown).	The node remains, but all visible methods called directly or indirectly by the method are hidden.
Restore a hidden subgraph	Click the child navigator (default after a Hide Subgraph action).	Restores the visible methods in the previously hidden subgraph.

Navigating Calling Methods

Use the parent navigator to hide and show the methods that called a selected method. The term “parent navigator” refers to the triangle displaying a plus (+) or minus (-) sign at the left side of a node.

The following table lists the parent-related tasks you can perform.

Parent Navigator Task	Action	Response
Show only the subgraph for the selected method (and hide all other nodes)	Click the parent navigator (default action if other subgraphs are shown).	The method is selected and all direct (children) and indirect (descendants) calls to methods are displayed. All other nodes are hidden.
Restore nodes removed by showing only a subgraph	Click the parent navigator (default after a Show Only Subgraph action).	The parent methods and their trees are displayed.

Going Back to Previous Views of the Graph

JProbe keeps a history of your hide and show actions as graph views. As in a Web browser, you can go back and forth in the view history using the **Back** and **Forward** toolbar buttons.

Customizing the Call Graph

You can change how time is displayed in the method list (**Tools** > **Edit Options** > **Data Display** > **Performance**). You can also customize your view of the Methods view by changing the font family, size, and style, as well as the color scheme.

- Changing the Scale Gradation
- Changing the Color Scheme
- Customizing Fonts

Changing the Scale Gradation

By default, the scale gradation is geometric, which means that (except for the first two ranges) the range of the segments doubles from minimum to maximum. For example, if the first segment spans 15 units, the second spans 15, the third 30, the fourth 60, and so on, doubling in size until the maximum value is reached. Use a geometric gradation to see the relative relationships among expensive nodes in the graph, in particular when a graph has a few nodes with values that far exceed the other nodes.

There will be times, however, when you want a quantitative view of the graph. In this case, you can change the scale gradation to linear; the color scale is divided into segments of equal size, and the graph highlights the few, most expensive nodes.

To select the gradation used for the scale:

- To highlight quantitative differences among the nodes, right-click the color scale and select **Linear Mapping**.
- To highlight relative relationships among the nodes, right-click the color scale and select **Geometric Mapping** (default).

Changing the Color Scheme

You can select from a few predefined color schemes. In all schemes, the bright color represents the most expensive methods while the dull color is for the least expensive

method. Select a color scheme that suits your preference. If you want to print a graph in black and white, the best scheme to use is **Blue to Light Gray**.

To change the color scheme:

- Right-click the color scale and select a color scheme.

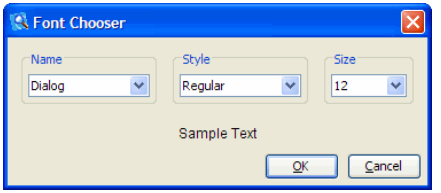
Customizing Fonts

Use the Font Chooser dialog box to change the font family, size, and style of text in the graph.

To change fonts:

1. On the toolbar, click  to open the toolbar menu and select **Set Font**.

   The *Font Chooser* dialog box opens.



2. Select a new font from the **Name** list.
3. Select Regular, Bold, or Italic from the **Style** list.
4. Select a font size from the **Size** list.

   Your choices are displayed in the sample text below the list boxes.
5. Click **OK**.

Exploring the Method Detail View

The Calling Context Tree (CCT) is created using information captured during the execution of your application program. The CCT contains information about the relationships between data, that is, data in the call history. Using the information from the CCT, Performance can display the relationships between methods and how these relationships may affect the overall running of the application. For more information, see “Understanding Performance Data” on page 222.

When using Performance to profile your program data, you might want to view your data at different levels of abstraction. The Call Graph provides you with a view of methods in your code at a higher level of abstraction. If you are interested in investigating a single method and viewing the associations with its parent and children methods, you can view this data in the Method Detail view. This level of abstraction can provide valuable information especially when you are investigating methods that take a substantial amount of time to run.

This Method Detail view features are described in the following sections:

- Opening the Method Detail View
- Understanding the Method Detail View
- Navigating the Method Detail View
- Customizing the Method Detail View

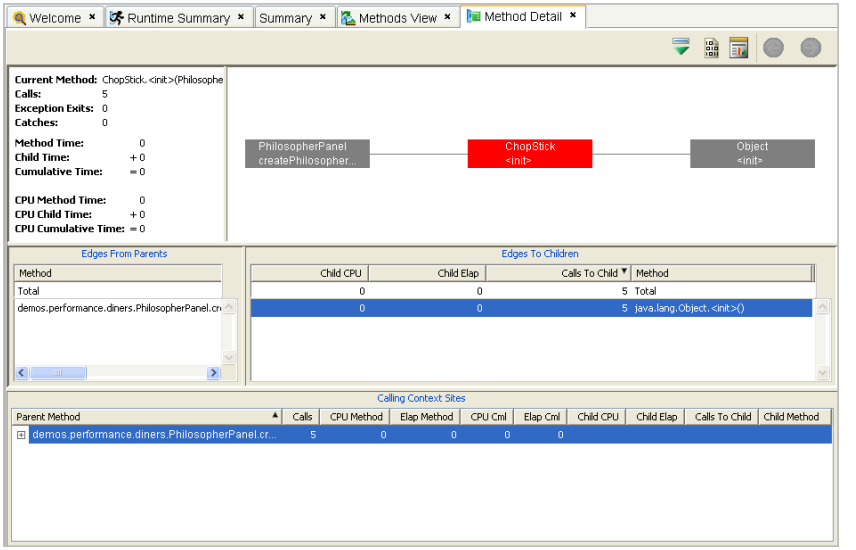
Opening the Method Detail View

The Method Detail view is only available by selecting a method in the Methods view.

To open the Method Detail view:

- In the Methods view, select a method, right-click, and select **Open Method Detail View** .

  The Method Detail view opens.



Understanding the Method Detail View

The Method Detail view displays method information similarly to the Methods view, but only for one method. There are five different areas in the view:

- Summary
- Graph
- Edges From Parents
- Edges To Children
- Calling Context Sites

The Methods view displays a graphical representation of the execution of your program. From here you can drill down on a specific method and view the results in the Method

Detail view. For this reason, it is important that you understand how your program works so that you understand the data presented in the methods and Method Detail views.

The Methods view displays some information that is captured in the CCT. To obtain information for a particular call, you hover over an edge between two nodes and a pop-up dialog appears containing the method name and data about the call. Alternatively the Method Detail view also displays the method names, and all of the information is displayed in a summary area. You can also view additional information about the individual nodes in the CCT table in the Context Calling Sites (CCS). For more information, see Building a Calling Context Tree.

Summary

The top left pane displays summary information about the method that you are viewing.

The **Show Source**  button allows you to view the code for the method.

Summary Item	Description
Current Method	The name of the method.
Calls	Number of calls to the method.
Exception Exits	The number of times the method exited due to an uncaught exception rather than a normal return.
Catches	The number of exceptions caught by the method.
Method Time	The amount of time that it took for the method to execute.
Child Time	The amount of time that it took for all children to execute.
Cumulative Time	The combination of method time and child time.
CPU Method Time	The CPU resources used by the method to execute.
CPU Child Time	The CPU resources used by all children to execute.

Summary Item	Description
CPU Cumulative Time	The combination of CPU method time and child time.
Recursive Calls	Number of recursive calls.
Recursive Time	Elapsed recursive time.
CPU Recursive Time	CPU recursive time.

Graph

The graph area displays the method that you are interested in (in red), and also up to five parent methods (located to the left of the current method) and associated child methods, (located to the right of the current method). Detailed information is shown in the *Edges From Parents* and *Edges To Children* panes.

Edges From Parents

The *Edges From Parents* pane displays information about the current method that you are investigating with respect to each of its parents. This information is directly related to the information held in the CCT. The parents contribute to the overall value of the method that you are investigating. Each row shows the contribution of a particular parent. The data corresponds to what you might see if you hovered over the edge from the named parent to the current method in the Methods view.

The following table lists information presented in the *Edges From Parents* table.

Column Data	Description
Method	The name of the parent method that called the current method. This column may contain recursion indicators  . A recursive indicator can appear in this column if the parent is a recursive caller, that is, if it is the parent of one or more recursive nodes. Cumulative time is not included in recursive nodes.
Calls	Displays the number of times that the parent called the current method.
CPU Method	The CPU resources used by the current method when called by its parent.
Elap Method	Displays the method time for the current method when called by its parent.
CPU Cml	Displays CPU cumulative time when called by parent.
Elap Cml	Displays cumulative time when called by parent.

Note In the Total row, Calls, Elap Cml, and Elap Method data match the information displayed in the Summary pane (respectively, Call, Cumulative Time, and Method Time).

Edges To Children

The *Edges To Children* area displays information about the current method's calls to its children.

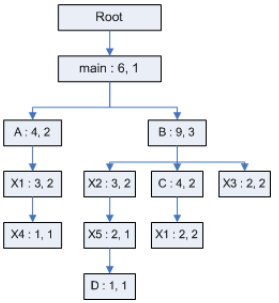
Column Data	Description
Child CPU	The contribution by the child method to the child CPU time of the current method.
Child Elap	The contribution by the child method to the child elapsed time of the current method.
Calls To Child	The number of calls from the current method to the child method. This column may contain recursion indicators.
Method	The name of the child method called by the current method. This column may also contain one of the following recursive indicators:  Child is a recursive node. Navigating to this child by double-clicking takes you to a method that is located up the calling tree from the current method.  Child is on path to a recursive node for current method. Child Time will not include time from the recursive node.  Both of the above.

Note In the Total row, the total Child CPU and Child Elap data matches the information displayed in the Summary pane (CPU Child Time and Child Time, respectively).

Calling Context Sites

The Calling Context Sites (CCS) table shows individual CCT nodes. For more information about the CCT see, "Understanding Performance Data" on page 222. It allows you to inspect the CCT directly, instead of viewing data through the Call Graph. The action of clicking a parent or child from the edges tables populates the CCS table

with CCT nodes that have a parent or child. The CCS is a tree table with child nodes that are collapsed. Click the + to expand the children.



Simple CCT

For example, the Call Graph can only show you aggregated data for `X1`. But if you view the method in the Method Detail view we can see the separate `X1` nodes: if we select in the Edges From the Parents table, the Calling Context Sites table shows a single row, with 2 calls and method time of 3, and we can see it called `X4`. This is the left-most node, under `A`. If we select C from Parents, we see the other node, showing 2 calls and a method time of 2, and we can see that it never called `X4`.

Column Data	Description
Parent Method	The name of the parent method for the CCT node.
Calls	The number of times the parent called the current method for a CCT node.
CPU Method	The CPU resources used by the current method when called by its parent.
Elap Method	Displays the method time for the current method when called by its parent.
CPU Cml	Displays CPU cumulative time when called by parent.
Elap Cml	Displays cumulative time when called by parent.

Column Data	Description
Child CPU	The contribution by the child method to the child CPU time of the current method.
Child Elap	The contribution by the child method to the child elapsed time of the current method.
Calls to Child	Displays the number of calls to the current method by the parent method.
Child Method	The name of the child method.

For more information about recursive calls, see “Recursive Methods” on page 232.

Navigating the Method Detail View

Use the Method Detail view to navigate through the call stack and view the current method’s performance data. The Method Detail view behaves similarly to the history navigation of a web browser. You can use the forward button to move to the next method and use the back button to view previous method detail method. Double-click the parent in the Edges From Parents, or child in Edges To Children, to make that method the current method.

The following table lists the navigation tasks:

Task	Action	Response
Focus on call path to current method from a parent	Select parent in Edges From Parent.	The CCT nodes for the current method which have the selected parent are displayed in the CCS table.
Focus on a call path from current method to a child	Select child in Edges To Children.	The CCT nodes for the current method which call selected child nodes are displayed in the CCS table.

Task	Action	Response
Change the current method	Double-click the data in Edges From Parent or Edges To Children.	Changes the current method back to the selected method.
Move back in view history	Use **Back**  .	Changes the current method back to the previously viewed method.
Move forward in view history	Use **Forward**  .	Change the current method to the next method.
View source code for the current method	Click **Source**  to navigate to the source code (when available) of the selected method.	Opens the Source view for current method or to the Please Find Source dialog.

Customizing the Method Detail View

The data displayed in the Edges From Parents, Edges To Children, or Calling Context Sites tables provides you with a better understanding about methods and those methods that are deemed expensive. You can customize each tab to display data that is relevant to you.

Task	Action	Response
Show/hide table columns	Right-click and select **Show/ Hide Columns**. Select the columns to show and click **OK**.	The selected columns are displayed. The rest are hidden.

Task	Action	Response
Resize column widths	To set the column width, drag the edge of the column heading left or right. To see all data in a column, right-click in the column and select **Size <column name> to Fit**. To fit all columns in the viewing area, right-click and select **Size All Columns to Fit**.	The columns are resized.
Sort the table by a metric	Click the column heading for the metric.	The table is sorted. To reverse the sort order, click the heading again.
Export the table data to CSV	Right-click in the table and select **Export to CSV**. Type a file name and click **Save**.	The file contains the table data separated by a comma. It may also contain headers.

Exploring the Performance Source View

When you open a performance snapshot in the Source view, you see your source code with line numbers. If you collected data at line granularity for any methods in this source code, you will also see columns for performance data by line. Available metrics for line-level data include Calls, Elapsed Method, and Elapsed Cml (for the line).

Note Line data is not shown for object and class initialization methods.

This section describes the features of the Source view:

- Opening the Source View
- Interacting with the Source View

Opening the Source View

If you have the source code for a method, you can open the Source view from a context menu or from the toolbar.

To open the Source view:

1. Select the method (that is, a row) in the Methods view or Method Detail view.
2. Right-click and select **Show Source**.

   or

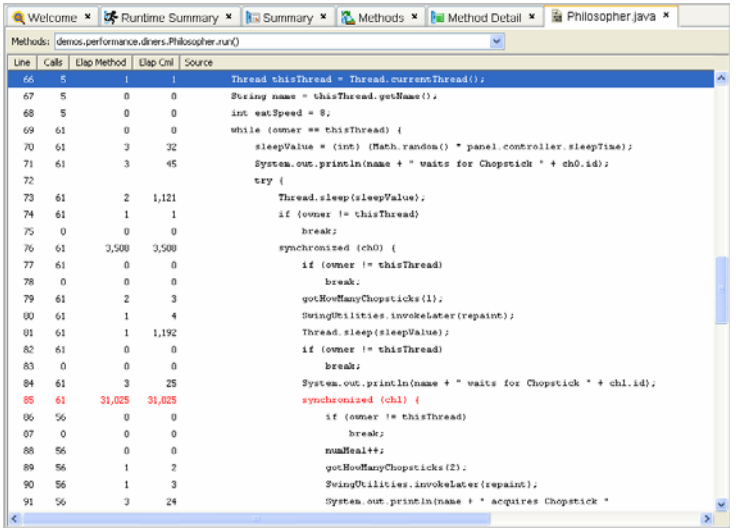
   Click **Show Source**  .

   or

   Right-click and select **Show Line Data** (if the selected method has line data collected).

   The Source view opens with the code for the selected method displayed.

   If you collected your data using Line Level filters, the Source view also displays line level detail in additional table columns.


Welcome
Runtime Summary
Summary
Methods
Method Detail
Philosopher.java
Methods: demos.performance.diners.Philosopher.run()
Line Calls Elap Method Elap Cml Source
66 5 1 1 Thread thisThread = Thread.currentThread();
67 5 0 0 String name = thisThread.getName();
68 5 0 0 int eatSpeed = 8;
69 61 0 0 while (owner == thisThread) {
70 61 3 32 sleepValue = (int) (Math.random() * panel.controller.sleepTime);
71 61 3 45 System.out.println(name + " waits for Chopstick " + ch0.id);
72 try {
73 61 2 1,121 Thread.sleep(sleepValue);
74 61 1 1 if (owner != thisThread)
75 0 0 0 break;
76 61 3,508 3,508 synchronized (ch0) {
77 61 0 0 if (owner != thisThread)
78 0 0 0 break;
79 61 2 3 gotHowManyChopsticks(1);
80 61 1 4 SwingUtilities.invokeLater(repaint);
81 61 1 1,192 Thread.sleep(sleepValue);
82 61 0 0 if (owner != thisThread)
83 0 0 0 break;
84 61 3 25 System.out.println(name + " waits for Chopstick " + ch1.id);
85 61 31,025 31,025 synchronized (ch1) {
86 56 0 0 if (owner != thisThread)
87 0 0 0 break;
88 56 0 0 numMeal++;
89 56 1 2 gotHowManyChopsticks(2);
90 56 1 3 SwingUtilities.invokeLater(repaint);
91 56 3 24 System.out.println(name + " acquires Chopstick "

Interacting with the Source View

If you collected line information for some or all of the methods in this source code, you can jump to a specific method in the code by selecting the method name from the Methods list.

You can also sort the columns of data in the right pane.

Task	Action	Response
Show/hide table columns	Right-click and select **Show/ Hide Columns**. Select the columns to show and click **OK**.	The selected columns are displayed. The rest are hidden.

Task	Action	Response
Resize column widths	To set the column width, drag the edge of the column heading left or right. To see all data in a column, right-click in the column and select **Size <column name> to Fit**. To fit all columns in the viewing area, right-click and select **Size All Columns to Fit**.	The columns are resized.
Rearrange columns	Drag a column by its heading and drop it in a new location.	The column appears in its new location.
Sort the table by a metric	Click the column heading for the metric.	The table is sorted. To reverse the sort order, click the heading again.
Export the table data to CSV	Right-click in the table and select **Export to CSV**. Type a file name and click **Save**.	The file contains the table data separated by a comma. It may also contain headers.

Exploring the Performance Difference View

The Performance Difference view lists the differences between the current snapshot and a previous (baseline) snapshot. To prepare to use this tab, you take a snapshot of your heap data before implementing changes to fix a performance bottleneck. After you have solved a performance bottleneck, you can rerun the application, take a snapshot, and compare this new snapshot to the old snapshot to see exactly where you achieved performance gains.

Caution To get meaningful results, you have to compare similar snapshots. This means you need to use the same session settings and take the snapshots at about the same point in each session.

This section describes the features of the Performance Difference view:

- Opening the Performance Difference View
- Understanding the Performance Difference View
- Interacting with the Performance Difference View

Opening the Performance Difference View

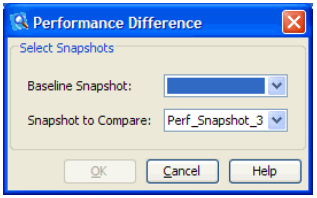
Before opening the Snapshot Difference view, you should load the performance snapshots you want to compare. For details, see “Managing Snapshots” on page 447.

Caution Make sure that you selected valid snapshots — you cannot compare a snapshot against itself or against a snapshot from a different application.

To open the Performance Difference view:

1. Load the snapshots that you want to compare.
2. In the Snapshots Navigator, right-click a snapshot and select **Snapshot Differencing**.

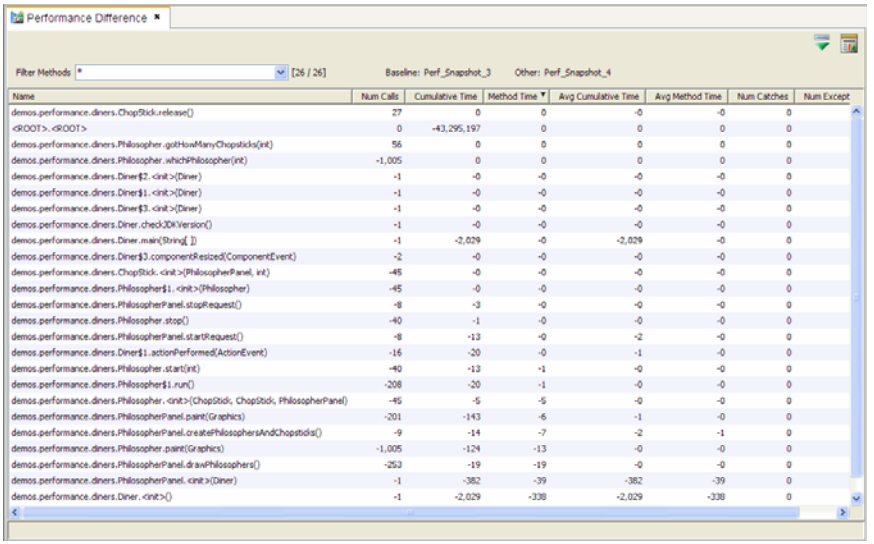
   The Performance Difference dialog box opens. The snapshot you selected is displayed in the *Snapshot to Compare* field.



3 Select another snapshot from the *Baseline Snapshot* list.

4 Click **OK**.

The Performance Difference view opens.


Performance Difference
Filter Methods *
[26 / 26]
Baseline: Perf_Snapshot_3
Other: Perf_Snapshot_4
Name
Num Calls
Cumulative Time
Method Time
Avg Cumulative Time
Avg Method Time
Num Catches
Num Except
demos.performance.diners.ChopStick.release() 27 0 0 -0 -0 0
<ROOT>.<ROOT> 0 -43,295,197 0 0 0 0
demos.performance.diners.Philosopher.gotHowManyChopsticks(int) 56 0 0 0 0 0
demos.performance.diners.Philosopher.whichPhilosopher(int) -1,005 0 0 0 0 0
demos.performance.diners.Diner$2.<init>(Diner) -1 -0 -0 -0 -0 0
demos.performance.diners.Diner$1.<init>(Diner) -1 -0 -0 -0 -0 0
demos.performance.diners.Diner$3.<init>(Diner) -1 -0 -0 -0 -0 0
demos.performance.diners.Diner.checkJDKVersion() -1 -0 -0 -0 -0 0
demos.performance.diners.Diner.main(String[ ]) -1 -2,029 -0 -2,029 -0 0
demos.performance.diners.Diner$3.componentResized(ComponentEvent) -2 -0 -0 -0 -0 0
demos.performance.diners.ChopStick.<init>(PhilosopherPanel, int) -45 -0 -0 -0 -0 0
demos.performance.diners.Philosopher$1.<init>(Philosopher) -45 -0 -0 -0 -0 0
demos.performance.diners.PhilosopherPanel.stopRequest() -8 -3 -0 -0 -0 0
demos.performance.diners.Philosopher.stop() -40 -1 -0 -0 -0 0
demos.performance.diners.PhilosopherPanel.startRequest() -8 -13 -0 -2 -0 0
demos.performance.diners.Diner$1.actionPerformed(ActionEvent) -16 -20 -0 -1 -0 0
demos.performance.diners.Philosopher.start(int) -40 -13 -1 -0 -0 0
demos.performance.diners.Philosopher$1.run() -208 -20 -1 -0 -0 0
demos.performance.diners.Philosopher.<init>(ChopStick, ChopStick, PhilosopherPanel) -45 -5 -5 -0 -0 0
demos.performance.diners.PhilosopherPanel.paint(Graphics) -201 -143 -6 -1 -0 0
demos.performance.diners.PhilosopherPanel.createPhilosophersAndChopsticks() -9 -14 -7 -2 -1 0
demos.performance.diners.Philosopher.paint(Graphics) -1,005 -124 -13 -0 -0 0
demos.performance.diners.PhilosopherPanel.drawPhilosophers() -253 -19 -19 -0 -0 0
demos.performance.diners.PhilosopherPanel.<init>(Diner) -1 -382 -39 -382 -39 0
demos.performance.diners.Diner.<init>() -1 -2,029 -338 -2,029 -338 0

The values in the table represent the differences between the snapshots. A negative value shows a reduction in time spent or objects created, which translates to a performance improvement.

Understanding the Performance Difference View

When you selected snapshots to compare, you selected snapshots that represented similar session settings and were taken at approximately the same time in the session. Also, make sure that you have made the same edits (such as prunes and isolates) in both snapshots. For more information, “Editing the Data Set” on page 318.

The Filter Methods fields enables you to filter data based on the package or class of a selected method. By filtering methods, you view only the methods that you are interested in investigating.

Caution Large percentage changes in methods with short method times may not be meaningful, as even normal fluctuation in times may be seen as a large percentage increase or decrease.

The difference values are the result of a subtraction: the compared snapshot minus the baseline snapshot. A negative value represents a performance gain, that is, it means less time was spent and/or fewer objects were created in the compared snapshot. A positive value means more time was spent or objects created. A value of zero means no change. If you have percentages turned on, you can see how great a change occurred.

The percentages are calculated using the following formula:

```
Percentage = (Compared - Baseline) / Baseline * 100
```

If the resulting value is >1000% or <-1000%, the change is reported as “huge.” If a method was not called in the baseline snapshot but is called in the compared snapshot, the percentage is replaced with a plus sign (+) to indicate a new method.

Interacting with the Performance Difference View

You interact with this view in the same way as with the Method List in the Methods view. You can show metric columns, sort the table by a metric column, use the Find feature, and use the Filter feature. For information on how to use the Performance Difference view, see “Measuring Performance Improvements” on page 258.

The following table lists the tasks that you can perform:

Task	Action	Response
Filter methods by package	Select a method, right-click, and select **Set Filter Based On > Method Package**. or Type or select a filter in the *Filter Methods* field.	The table refreshes to show only the methods in that package.

Task	Action	Response
Filter methods by class	Select a method, right-click, and select **Set Filter Based On** > **Method Class**. or Type or select a filter in the *Filter Methods* field.	The table refreshes to list only the methods in that class.
Show/hide table columns	Right-click and select **Show/ Hide Columns**. Select the columns to show and click **OK**.	The selected columns are displayed. The rest are hidden.
Resize column widths	To set the column width, drag the edge of the column heading left or right. To see all data in a column, right-click in the column and select **Size <column name> to Fit**. To fit all columns in the viewing area, right-click and select **Size All Columns to Fit**.	The columns are resized.
Rearrange columns	Drag a column by its heading and drop it in a new location.	The column appears in its new location.
Sort the table by a metric	Click the column heading for the metric.	The table is sorted. To reverse the sort order, click the heading again.
Export the table data to CSV	Right-click in the table and select **Export to CSV**. Type a file name and click **Save**.	The file contains the table data separated by a comma. It may also contain headers.
Create a report for this view	Click **Export Data to HTML**  .	The Create Performance Difference Report dialog box appears.

Setting Preferences for Performance Views

You set data display options in the Edit Options dialog box.

Option	Description
Show Percentage Time Values in Methods View Trees	Defines if the results are displayed in the Methods view as percentages or as actual time values.
Time Unit	Defines the unit for measuring time intervals displayed in Performance views. The following options are available: • Nanoseconds • Microseconds • Milliseconds (the default) • Seconds
Time Precision	Defines the precision used when calculating the time units displayed in Performance views. The following options are available: no decimal places or one, two, three, four, or five decimal places.
JDBC Truncation Style	Defines the JDBC truncation method. The following options are available: • Truncate at the Beginning of Connection and Statement Strings • Truncate in the Middle of Connection and Statement Strings • Truncate at the End of Connection and Statement Strings (default) • Do Not Truncate Connection and Statement Strings

For information about setting other preferences in JProbe, see “Setting Preferences” on page 453.

13

Reporting Performance Analysis Results

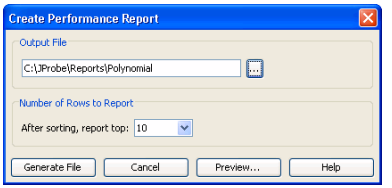
You can generate reports based on some of the Performance views. This chapter presents the Performance reports. The steps for creating a report are described under “Generating HTML Reports” on page 440.

This chapter contains the following sections:

Setting Report Options 344
Runtime Summary Report 344
Summary Report 345
Methods Report 346
Method Detail Report 347
Performance Difference Report 347

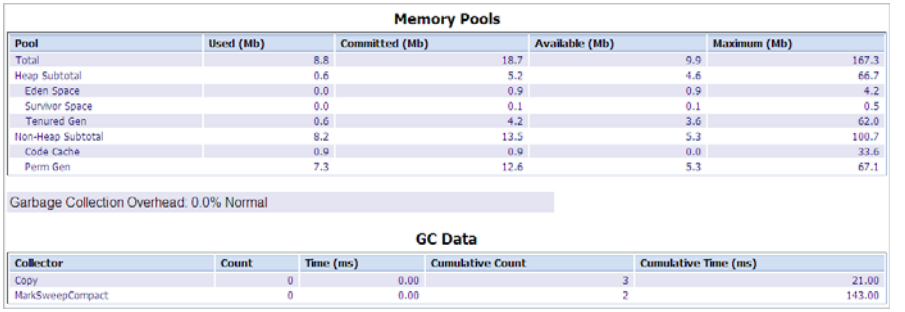
Setting Report Options

When you generate a Performance report, the number of methods in the report is controlled by the Number of Rows to Report option in the Create Performance Report dialog box. “Report the top 10” is the default option.



Runtime Summary Report

This report contains the content of the **Memory Pools** and **GC Data** tabs in the Runtime Summary view.



Memory Pools

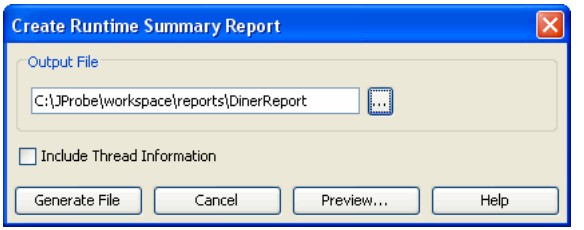
Pool	Used (Mb)	Committed (Mb)	Available (Mb)	Maximum (Mb)
Total	8.8	18.7	9.9	167.3
Heap Subtotal	0.6	5.2	4.6	66.7
Eden Space	0.0	0.9	0.9	4.2
Survivor Space	0.0	0.1	0.1	0.5
Tenured Gen	0.6	4.2	3.6	62.0
Non-Heap Subtotal	8.2	13.5	5.3	100.7
Code Cache	0.9	0.9	0.0	33.6
Perm Gen	7.3	12.6	5.3	67.1

Garbage Collection Overhead: 0.0% Normal

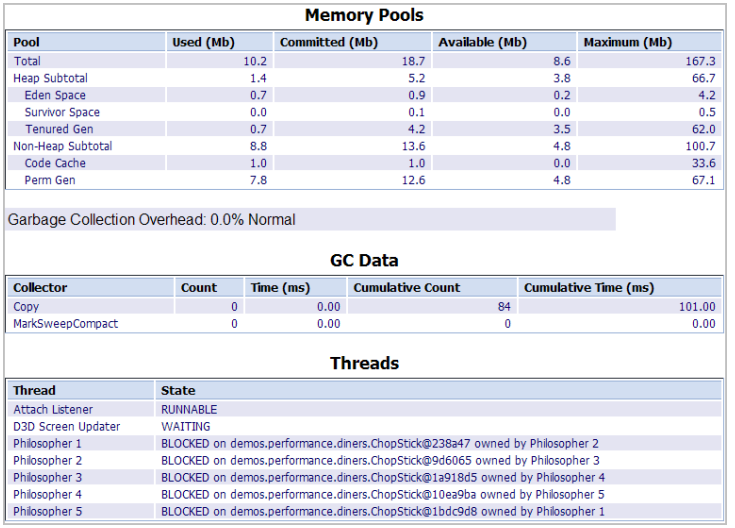
GC Data

Collector	Count	Time (ms)	Cumulative Count	Cumulative Time (ms)
Copy	0	0.00	3	21.00
MarkSweepCompact	0	0.00	2	143.00

If you select the *Include Thread Information* check box in the Runtime Summary Report dialog box, this report also includes the states of all threads currently displayed in the **Threads** tab.



The list includes the identity of any locks these threads are blocked on, but no category entries are reported.



Memory Pools

Pool	Used (Mb)	Committed (Mb)	Available (Mb)	Maximum (Mb)
Total	10.2	18.7	8.6	167.3
Heap Subtotal	1.4	5.2	3.8	66.7
Eden Space	0.7	0.9	0.2	4.2
Survivor Space	0.0	0.1	0.0	0.5
Tenured Gen	0.7	4.2	3.5	62.0
Non-Heap Subtotal	8.8	13.6	4.8	100.7
Code Cache	1.0	1.0	0.0	33.6
Perm Gen	7.8	12.6	4.8	67.1

Garbage Collection Overhead: 0.0% Normal

GC Data

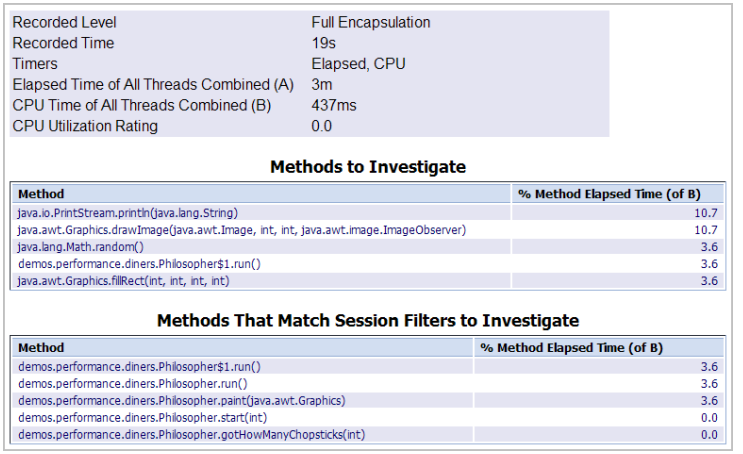
Collector	Count	Time (ms)	Cumulative Count	Cumulative Time (ms)
Copy	0	0.00	84	101.00
MarkSweepCompact	0	0.00	0	0.00

Threads

Thread	State
Attach Listener	RUNNABLE
D3D Screen Updater	WAITING
Philosopher 1	BLOCKED on demos.performance.diners.ChopStick@238a47 owned by Philosopher 2
Philosopher 2	BLOCKED on demos.performance.diners.ChopStick@9d6065 owned by Philosopher 3
Philosopher 3	BLOCKED on demos.performance.diners.ChopStick@1a918d5 owned by Philosopher 4
Philosopher 4	BLOCKED on demos.performance.diners.ChopStick@10ea9ba owned by Philosopher 5
Philosopher 5	BLOCKED on demos.performance.diners.ChopStick@1bdc9d8 owned by Philosopher 1

Summary Report

This report contains the content of the Summary view.



Recorded Level	Full Encapsulation
Recorded Time	19s
Timers	Elapsed, CPU
Elapsed Time of All Threads Combined (A)	3m
CPU Time of All Threads Combined (B)	437ms
CPU Utilization Rating	0.0

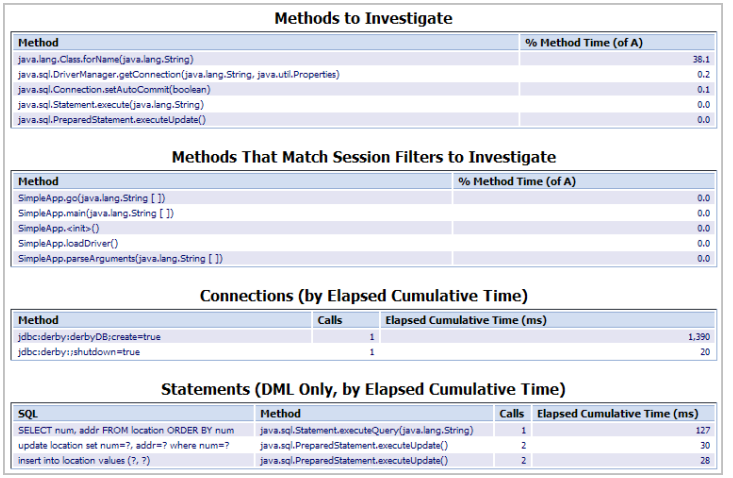
Methods to Investigate

Method	% Method Elapsed Time (of B)
java.io.PrintStream.println(java.lang.String)	10.7
java.awt.Graphics.drawImage(java.awt.Image, int, int, java.awt.image.ImageObserver)	10.7
java.lang.Math.random()	3.6
demos.performance.diners.Philosopher$1.run()	3.6
java.awt.Graphics.fillRect(int, int, int, int)	3.6

Methods That Match Session Filters to Investigate

Method	% Method Elapsed Time (of B)
demos.performance.diners.Philosopher$1.run()	3.6
demos.performance.diners.Philosopher.run()	3.6
demos.performance.diners.Philosopher.paint(java.awt.Graphics)	3.6
demos.performance.diners.Philosopher.start(int)	0.0
demos.performance.diners.Philosopher.gotHowManyChopsticks(int)	0.0

When the JDBC information is present, the Performance Summary report also includes the connections and SQL statement sections.



Methods to Investigate

Method	% Method Time (of A)
java.lang.Class.forName(java.lang.String)	38.1
java.sql.DriverManager.getConnection(java.lang.String, java.util.Properties)	0.2
java.sql.Connection.setAutoCommit(boolean)	0.1
java.sql.Statement.execute(java.lang.String)	0.0
java.sql.PreparedStatement.executeUpdate()	0.0

Methods That Match Session Filters to Investigate

Method	% Method Time (of A)
SimpleApp.go(java.lang.String [ ])	0.0
SimpleApp.main(java.lang.String [ ])	0.0
SimpleApp.<init>()	0.0
SimpleApp.loadDriver()	0.0
SimpleApp.parseArguments(java.lang.String [ ])	0.0

Connections (by Elapsed Cumulative Time)

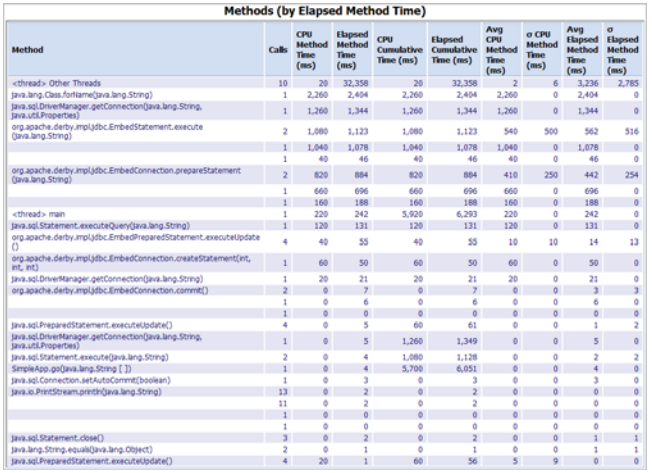
Method	Calls	Elapsed Cumulative Time (ms)
jdbc:derby:derbyDB;create=true	1	1,390
jdbc:derby:;shutdown=true	1	20

Statements (DML Only, by Elapsed Cumulative Time)

SQL	Method	Calls	Elapsed Cumulative Time (ms)
SELECT num, addr FROM location ORDER BY num	java.sql.Statement.executeQuery(java.lang.String)	1	127
update location set num=?, addr=? where num=?	java.sql.PreparedStatement.executeUpdate()	2	30
insert into location values (?, ?)	java.sql.PreparedStatement.executeUpdate()	2	28

Methods Report

This report contains the content of the lower table in the Methods view. Before generating this report, sort the table by the metric in which you are interested.

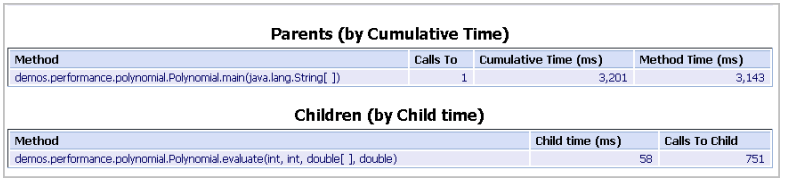


Methods (by Elapsed Method Time)

Method	Calls	CPU Method Time (ms)	Elapsed Method Time (ms)	CPU Cumulative Time (ms)	Elapsed Cumulative Time (ms)	Avg CPU Method Time (ms)	σ CPU Method Time (ms)	Avg Elapsed Method Time (ms)	σ Elapsed Method Time (ms)
<thread> Other Threads	10	20	32,358	20	32,358	2	6	3,236	2,785
java.lang.Class.forName(java.lang.String)	1	2,260	2,404	2,260	2,404	2,260	0	2,404	0
java.sql.DriverManager.getConnection(java.lang.String, java.util.Properties)	1	1,260	1,344	1,260	1,344	1,260	0	1,344	0
org.apache.derby.impl.jdbc.EmbedStatement.execute (java.lang.String)	2	1,080	1,123	1,080	1,123	540	500	562	516
	1	1,040	1,078	1,040	1,078	1,040	0	1,078	0
	1	40	46	40	46	40	0	46	0
org.apache.derby.impl.jdbc.EmbedConnection.prepareStatement (java.lang.String)	2	820	884	820	884	410	250	442	254
	1	660	696	660	696	660	0	696	0
	1	160	188	160	188	160	0	188	0
<thread> main	1	220	242	5,920	6,293	220	0	242	0
java.sql.Statement.executeQuery(java.lang.String)	1	120	131	120	131	120	0	131	0
org.apache.derby.impl.jdbc.EmbedPreparedStatement.executeUpdate ()	4	40	55	40	55	10	10	14	13
org.apache.derby.impl.jdbc.EmbedConnection.createStatement(int, int, int)	1	60	50	60	50	60	0	50	0
java.sql.DriverManager.getConnection(java.lang.String)	1	20	21	20	21	20	0	21	0
org.apache.derby.impl.jdbc.EmbedConnection.commit()	2	0	7	0	7	0	0	3	3
	1	0	6	0	6	0	0	6	0
	1	0	0	0	0	0	0	0	0
java.sql.PreparedStatement.executeUpdate()	4	0	5	60	61	0	0	1	2
java.sql.DriverManager.getConnection(java.lang.String, java.util.Properties)	1	0	5	1,260	1,349	0	0	5	0
java.sql.Statement.execute(java.lang.String)	2	0	4	1,080	1,128	0	0	2	2
SimpleApp.go(java.lang.String [ ])	1	0	4	5,700	6,051	0	0	4	0
java.sql.Connection.setAutoCommit(boolean)	1	0	3	0	3	0	0	3	0
java.io.PrintStream.println(java.lang.String)	13	0	2	0	2	0	0	0	0
	11	0	2	0	2	0	0	0	0
	1	0	0	0	0	0	0	0	0
	1	0	0	0	0	0	0	0	0
java.sql.Statement.close()	3	0	2	0	2	0	0	1	1
java.lang.String.equals(java.lang.Object)	2	0	1	0	1	0	0	1	1
java.sql.PreparedStatement.executeUpdate()	4	20	1	60	56	5	9	0	0

Method Detail Report

This report contains the content of the Edges From Parents and the Edges to Children tables in the Method Detail view. Before generating this report, sort the table by the metric in which you are interested.



Parents (by Cumulative Time)

Method	Calls To	Cumulative Time (ms)	Method Time (ms)
demos.performance.polynomial.Polynomial.main(java.lang.String[ ])	1	3,201	3,143

Children (by Child time)

Method	Child time (ms)	Calls To Child
demos.performance.polynomial.Polynomial.evaluate(int, int, double[ ], double)	58	751

Performance Difference Report

This report contains the content of the Performance Difference view. Before generating this report, sort the table by the metric in which you are interested.

Methods (by Method Time)

Method	Num Calls	Cumulative Time (ms)	Method Time (ms)	Avg Cumulative Time (ms)	Avg Method Time (ms)	Num Catches	Num Exception Exits
demos.performance.polynomial.Polynomial.main (java.lang.String[ ])	0	-2,982	0	-1,928	0	0	0
<ROOT>.<ROOT>	0	-2,982	0	0	0	0	0
demos.performance.polynomial.Polynomial.evaluate (int, int, double[ ], double)	0	-2,982	-3,101	-1,928	-2,106	0	0

14

Troubleshooting Performance Analysis Results

As you work with JProbe, you may need help to get the most out of your analysis sessions. This chapter presents a few tips for solving some Performance analysis problems and for running sessions efficiently. For Performance analysis issues that are not covered here, refer to Quest's technical support.

This chapter contains the following sections:

Performance Troubleshooting Tips....350
Performance Session Runtime Tips....351
Performance Help Messages....353

Performance Troubleshooting Tips

The following tips can help you with problems you may encounter. See the *ReleaseNotes.htm* file for other product-related issues.

I am having difficulties configuring my environment.

Contact Quest Support for assistance. When you do, be prepared to describe your environment. The customer service representative may ask you to run a JProbe session that includes some debugging options to describe your settings more fully.

My program called a method in a derived class, but I don't see the method in the Call Graph.

This issue arises when you set an encapsulated filter on the derived class, but use a method- or line-level filter on the base class. What happens is that when you call a method in the derived class through a reference of the base type, JProbe attributes the performance data for the derived method to the calling method. The easiest fix is to ensure that both classes use the same type of filter. Alternatively, you can set a line-level filter on the calling method to see how much time was spent calling the derived method.

Consider the following program snippet and imagine that `Base` has a method-level detail filter, while `Derived` uses an encapsulated filter.

```
class Base {
     public static void main(String args[]) {
          Base base = new Derived();
          base.method();
     }
      public void method() {
          System.out.println("In Base.method()");
     }
}
class Derived extends Base {
      public void method() {
           System.out.println("In Derived.method()");
     }
}
```

In this example, all performance data for `Derived.method()` is attributed to `main()`, and `Derived.method()` does not appear in the Call Graph or Method List. If you change the filter on the `Derived` class to method-level, the performance data will be correctly attributed to `Derived.method()`.

The session hangs when running an application server under load.

If you notice excessive heap growth without garbage collections followed by an application crash, increase the `-Xms`, `-Xmx` parameters or lighten the load so that the JVM has a chance to catch up and do a garbage collection. As the JVM caches data, you can increase the load.

For example, the compiling process for JSPs takes up a significant amount of memory. If your session is failing at a load of only, say, five pages per second, it is likely a memory allocation problem. Running through the JSPs at a slow pace allows the application server to cache them, and afterwards you can increase the load on the system without difficulty.

Active methods may report as `<unknown>.<unknown>()` after (re)starting data collection

When not recording, Performance does not keep track of where your program is executing. When you (re)start recording, it will track future calls, but it may still not know what was executing when the recording started, until it exits that method. If that method is a long-running method, for example a `run()` method on a Runnable, the method may still be unknown at the time of the snapshot.

To avoid this situation, start recording before the long-running method is started, for example by using a trigger on the method itself.

Performance Session Runtime Tips

The following tips will help you to improve session runtime performance.

Try different supported JVMs.

The session runtime performance depends in part on the speed at which your Java program runs. Modern JVMs, such as Java 6, generally improve program performance. For more information, see the JVM documentation and the *JProbe Installation Guide*.

Exit your program before investigating snapshots.

If you open a snapshot in one of the investigation tools while your program continues to run, you may find that the JProbe Console responds more slowly than desired, because it is competing for resources with the JProbe Analysis Engine. To improve JProbe Console performance, exit your program before opening a snapshot, or disconnect from the session (if you don’t want to end your application).

Choose not to monitor the session or monitor it remotely.

Monitoring the session requires that two JVMs be running: the JVM under analysis that executes your program and the JVM running the JProbe Console. If you monitor remotely, the JVMs are not competing for resources on the same computer. Alternatively, you can run a JProbe session without the JProbe Console. Instead of interacting with the session through the JProbe Console, you use triggers to collect data and take snapshots. After the session ends, you can open the snapshots in the JProbe Console.

Set filters to reduce the amount of data collected.

Filters reduce overhead and improve session performance significantly. The default filter setting collects data on methods in your package, and encapsulates data for the methods contained in all other packages. If you were to collect data on all methods, the session would run very slowly and the data collected would be of little value in tracking down performance bottlenecks. As you confirm which of your methods are working well, you can encapsulate those methods too. For more information, see “Encapsulating Data on Methods” in the *JProbe User Guide*.

Only collect line data on problem methods.

Only use the Line Level filter on methods that you have identified as performance bottlenecks. Collecting data line-by-line causes the session to run more slowly. For more information, see “Collecting Data on Lines of Code” in the *JProbe User Guide*.

Reduce the number of nodes in the Call Graph.

Displaying a large or complex Call Graph can take longer than desired. To improve the responsiveness of the Call Graph, use some of the following techniques to reduce the number of methods that need to be maintained:

- use data collection filters
- exclude threads
- prune subtrees
- isolate a subtree

Performance Help Messages

When you try to execute an action that is not enabled, the JProbe Console usually displays a message explaining why that action is currently disabled and suggests what you need to do to resolve the situation.

This section provides a list of memory help message you may encounter while doing a Performance Analysis using JProbe:

- "No Line Data" on page 353

No Line Data

You can view line-level details only if your snapshot contains line data.

To collect line data, select the *Line Level* option when defining your data collection filters, and run a new session to obtain a new snapshot.

For more information about line data, see "Metrics and Data Collection Filters" on page 241 and "Collecting Data on Lines of Code" on page 255.

If you have already set your data collection filters to collect line data and still get the `No Line Data` message, it may be that the class was compiled without line information. You may need to recompile the class with your compiler's debug option (for example, `-g` in `javac`).

15

Strategies for Coverage Analysis

The Coverage analysis tool provides a statement-level assessment of your test case coverage. It can also be configured to analyze how well your test cases cover the conditions in your source code. You do not need a specific build, a complete build, or access to all the source code to start analyzing your test case coverage. As long as you can compile the source code, you can use the Coverage tool.

Tip JProbe does not analyze loops or counter overflows.

Quality Assurance analysts use the JProbe Coverage tool to assess how much an application's code is covered by their test case or suite of test cases. Software developers can also use the Coverage tool to verify the completeness of their unit tests and to locate obsolete code.

This chapter provides recommandations for how to use the JProbe Coverage analysis tool in your development environment.

This chapter contains the following sections:

Assessing an Existing Test Suite 356
Determining Overall Coverage 356
Modifying a Test Suite 356
Creating a New Test Suite 356
Keeping Up with Development 357

Assessing an Existing Test Suite

If you have a large, complex, or long-running process, you can perform a preliminary analysis of your test suite at the method level. The sessions run quickly and you can clearly see which methods are missed entirely by your test cases. Once these areas have been identified and corrected, you can then follow up with more thorough line-by-line analyses to find missed lines of code.

In some cases, your method-level analysis may point out blocks of code that were not exercised because your test case did not follow a particular conditional branch. In this case, you should run the test cases again while tracking condition coverage.

Determining Overall Coverage

First you create a baseline snapshot that lists all the classes in your application. Then you run a Coverage analysis on each test case, being careful to use the same class files, coverage level (method or line), and filter settings as the baseline snapshot. When you merge the test case snapshots with the baseline snapshot, the merged snapshot tells you the percentage of overall coverage that your test suite provides. You can generate reports to communicate your results to management and the rest of your team.

Modifying a Test Suite

As you analyze your test suite, you may find areas of code that were not exercised. Sometimes it may not be clear if you need to test that code or not. To find out, you can generate reports and show them to the software developer responsible for the code. The software developer should be able to tell you if the code needs to be tested or if it is obsolete.

Creating a New Test Suite

If you are starting your test suite from scratch, a good first strategy is to talk to the software development team and determine which use cases are most attempted by your users. Create test cases for those use cases first and then use JProbe to ensure your test cases cover all the code affected by those use cases. This strategy ensures that, at a minimum, your test suite covers the critical areas of your program. It also helps to

reduce redundancy and makes the most of your testing effort. You can add additional tests and work toward overall coverage of your application as the product matures.

Keeping Up with Development

One of the pitfalls of parallel testing is that the code and the interface can change from day to day, making some of your test cases out-of-date before you can run them. By using JProbe, you can be confident your test suite keeps pace with programming changes. If you run a Coverage analysis daily and are familiar with results, JProbe can help you to identify changes to the code. The overall coverage percentage changes if new lines of code are added or removed. New lines of code are identified as missed in your results.

16

Running a Coverage Analysis

This chapter presents a basic analysis using default Coverage settings.

This chapter contains the following sections:

Setting Up JProbe for a Basic Coverage Analysis 360
Running Test Cases with JProbe 361
Investigating Your Results 363

Setting Up JProbe for a Basic Coverage Analysis

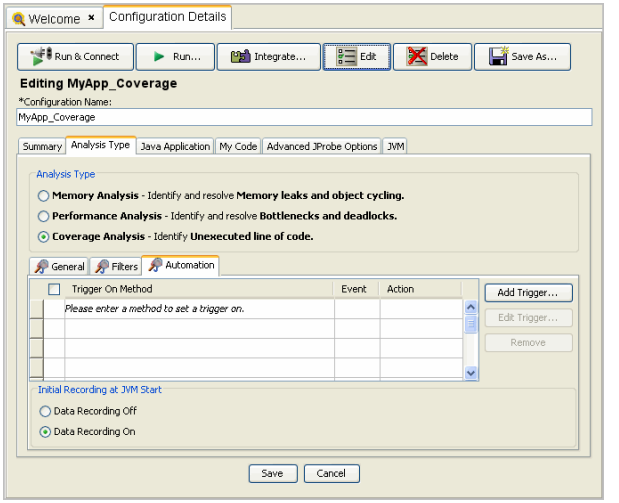
Note The following procedure assumes that you have already created a JProbe configuration. If you have not configured JProbe to run your application, see "Configuring JProbe to Analyze Your Application" on page 23.

To create a configuration for a Coverage analysis from an existing configuration:

1. Select the configuration in the Configurations Navigator.
2. Right-click the configuration name and select **Copy settings for Coverage Analysis**.

   A copy of the original configuration appears in the Configurations Navigator. The configuration settings are displayed in the **Configuration Details** tab.
3. To edit the new configuration, click **Edit**.

   The **Configuration Details** tab is placed in editing mode.



   a. Type in a name for this new configuration (for example, **MyApp_Coverage**).

   b. Edit the settings on any of the tabs, as necessary. A basic Coverage analysis uses the following default options:

- **Analysis Type > General** tab: Set **Coverage Level** to **Line**—JProbe tracks which lines of code in your application are called during your test case. Conditions are not tracked.
- **Analysis Type > Filters** tab: JProbe uses your application filters as the default data collection filters.
- **Analysis Type > Automation** tab: Set **Initial Recording at JVM Start** to **Data Recording On**.

Tip In future analyses, you may want to adjust these options. See "Analysis Options for Coverage" on page 368.

c Click **Save**.

4 To generate a startup script, click **Integrate**.

a Type in a location and file name for the settings file, and click **Save**.

b Click **Continue**.

c Type in a location and file name for the startup script, and click **Save**.

d Click **Close Wizard**.

Running Test Cases with JProbe

To begin the analysis, you will launch your application using the startup script created when you integrated your configuration. For more information, see "Setting Up JProbe for a Basic Coverage Analysis" on page 360.

This procedure assumes that you are running manual test cases or test cases through another tool. For Java SE applications, if you have an input file that you pass as a command line argument, you can edit your configuration to add the application argument. For more information, click Help in the **Java Application** tab.

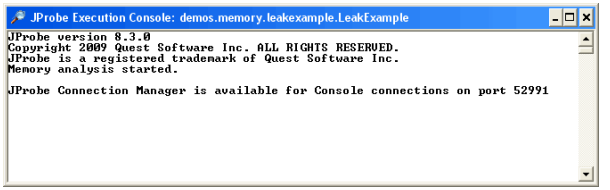
At the end of this procedure, you will have one or more Coverage snapshots. By default, a Coverage snapshot contains line-level data on classes in your application. Only classes that contain methods called during your test case are included in the snapshot.

Note This procedure assumes that you know how to start an application under JProbe. If not, see "Starting Your Application with JProbe Attached" on page 71.

To run a basic Coverage analysis:

1 Launch your application using the JProbe-generated startup script.

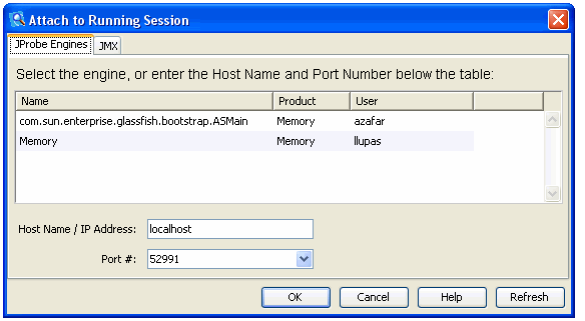
Caution For application servers, you may need to set environment variables and add command line arguments.


JProbe Execution Console: demos.memory.leakexample.LeakExample
JProbe version 8.3.0
Copyright 2009 Quest Software Inc. ALL RIGHTS RESERVED.
JProbe is a registered trademark of Quest Software Inc.
Memory analysis started.
JProbe Connection Manager is available for Console connections on port 52991

2 Launch the JProbe Console.

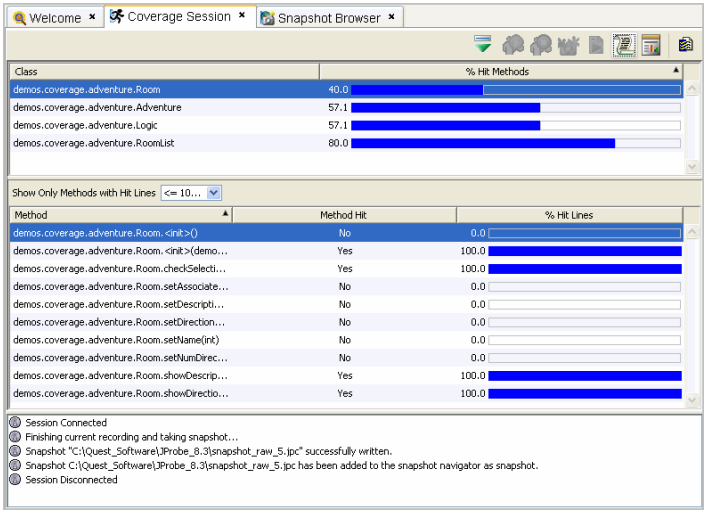
3 Click **Attach to Session** on the toolbar.

JProbe is searching for currently running engines and JVMs, and displays this list in the **Attach to Running Session** dialog box.



4 Select an engine from the list of currently running engines, and click **OK**.

The JProbe Console begins receiving data collected by the JProbe Analysis Engine. The **Coverage** tab is selected and the Coverage Session view is displayed.



The Coverage Session view shows data in terms of the number of methods and lines of code that were hit by your test case. For more information, see “Exploring the Coverage Session” on page 374.

5 Work through your test case as usual.

6 End your application as usual.

JProbe generates a Coverage snapshot that contains data on the test case.

7 Repeat for each test case in your test suite.

8 To make it easier to identify your snapshots, you may want to rename them.

- To rename and save the snapshot, right-click the snapshot and select **Save Snapshot As**. Save the snapshot with a meaningful name.
- To change the label only, right-click the snapshot and select **Change Label**.

Investigating Your Results

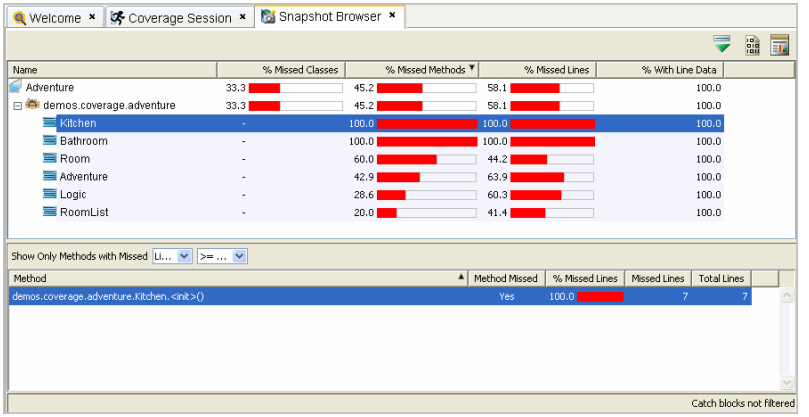
After you have run your test cases, you are ready to investigate your results. In the Snapshot Browser, JProbe displays results in terms of misses. A miss means that the code was not executed during your test cases.

The screen captures in the following procedure are taken from the Adventure tutorial. They are provided as an example only.

To investigate your results:

1. In the Snapshots Navigator, double-click a snapshot (original or merged).
2. In the Snapshot Browser, expand the package tree in the Application Tree pane (top pane).
3. In the **% Missed Methods** column, select a class that contains the highest percentage of missed methods. If you have to choose between classes with the same percentage, you may want to start with the one that contains the highest percentage of missed lines of code.

   The methods in the class are displayed in the Method Detail table (bottom pane). You can show more columns by right-clicking and selecting Show/Hide Columns.



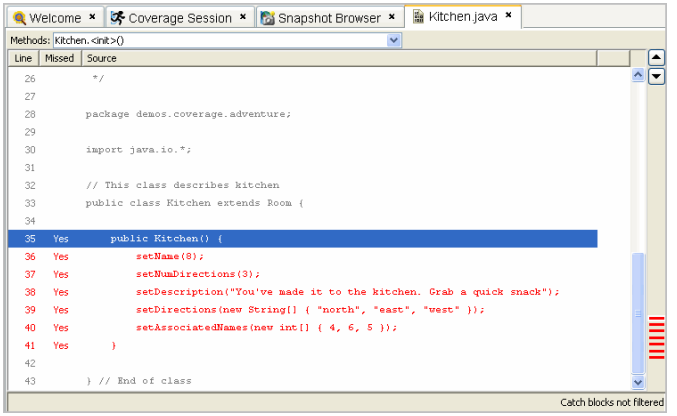
4. Select a method to investigate. A good first candidate would be an interesting method with a high number of missed lines.

   Tip Generally speaking, you can ignore the results for <init>() methods. If you were to examine the code, you would find that initializations (such as InputStream inputStream = null;) are reported as missed. This result is due to how the compiler identifies the lines in its line data table. For more information, see "Learning More About Coverage Results" on page 393.

5. To view the source code, double-click the method. You may be prompted to locate the source code file.

   If you have the source code, a Source view opens with the code displayed. By default, missed statements are displayed in red text and missed conditions (if tracked) in gold. Lines of code that were exercised by your test suite are in black

text. Lines with no data are in gray. If you are filtering out catch blocks, these are displayed in gray as well.



Along the right margin is a navigation aid called the Coverage Bar. The length of the Coverage Bar represents all the source code in the file, not just the code currently displayed in the view. The colored stripes on the bar represent the relative size and position of missed lines of code and missed conditions (if available). You can click a stripe to navigate to that block of code or use the arrows at the top of the bar to move to the previous or next missed code. For more information, see “Exploring the Coverage Source View” on page 384.

6 Determine if the missed lines of code in the method require testing and adjust your test cases. You may need to consult the software developer in charge of the code. If so, you can print a report of your results.

7 If you want to review all missed code in this source file, click the Next Stripe arrow on the Coverage Bar.

The view scrolls to show the next missed line.

8 To investigate a missed method in another source file, select the **Snapshot Browser** tab and repeat step 4 to step 7.

9 When you are done, rerun the session with the improved test cases.

Next Steps

As your program grows and evolves, your test suites need to keep pace with the changes. Changes are not always communicated to QA analysts. After you have achieved a satisfactory level of coverage, you can continue to use the JProbe Coverage analysis tool regularly. If your percentage of missed methods or missed lines of code

goes up, it is an indication that development has added new code that is not covered by your test cases.

17

Learning More About Coverage Options

In “Running a Coverage Analysis” on page 359, the default options were used. You can exercise more control over your analysis by experimenting with other options.

This chapter presents how to fine-tune the Coverage analysis tool and contains the following sections:

Analysis Options for Coverage 368

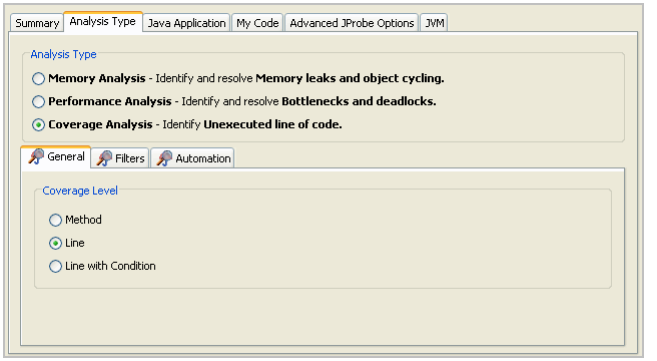
Analysis Options for Coverage

You can configure the following settings for your Coverage analysis:

- Coverage Options
- Filters for Coverage
- Triggers for Coverage

Coverage Options

The **Analysis Type > General** tab is where you can set the coverage level for a session. The coverage level determines the granularity of the data collected during the session.



The following table describes the Coverage Level options.

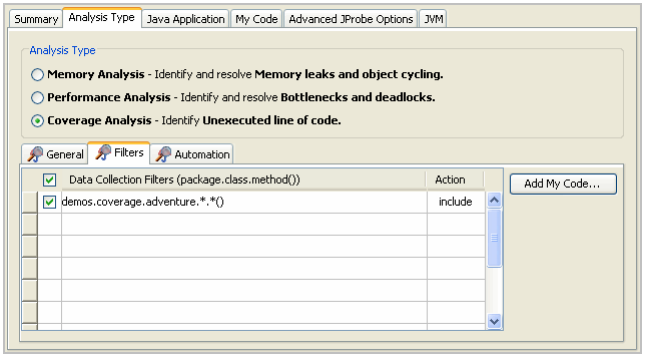
Coverage Level	Description
Method	JProbe tracks calls to the methods in your application, respecting any filters that are set. Method-level data is useful for a first pass analysis, especially of large or complex applications. You can use the results to identify methods that are missed entirely by your test case.

Coverage Level	Description
Line	JProbe tracks which lines of code are executed during the test case. You can review the results of a line-level analysis in the Source view. This is the default level. **Note** You need access to the source code for your application.
Line with Condition	JProbe collects line-level data plus it determines how many condition outcomes were exercised during the test case. Some of the conditions tracked by JProbe include: • `if...else` • `do...while` • `for... , and` • `switch...case...default` • `assert` statements

Filters for Coverage

The **Analysis Type > Filters** tab is where you can set filters for your Coverage analysis. Filters help you to focus on those aspects of the program that are important to you. Your Data Collection Filters table may already contain a set of default filters. The default filters are based on the description of your code that you provided (see the **My Code** tab in the Configuration wizard). You can use these filters or edit them.

For example, if you know one of your methods is 100% covered, you may choose to add a filter to stop collecting data about it. Consider adding filters whenever you have achieved the level of test case coverage you desire for a particular method.



Filters are defined in the same way for all the analysis tools, however the filter actions differ among the analysis tools. This section summarizes the filter actions for a Coverage analysis. To learn how to create filters, see “Refining Data Collection Filters” on page 422.

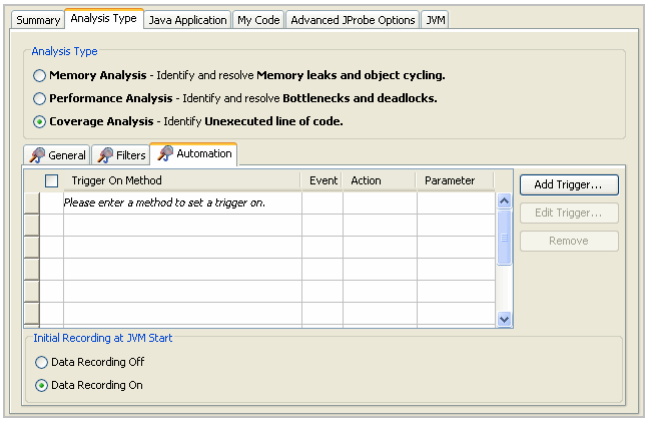
Filter Action	Description
Include	*Default.* By default, all classes and methods in the application (*except for native and abstract methods*) are included in the analysis and do not require explicit filters. You can use this action to include methods from a class that you have excluded. It also gives you the flexibility to restore a previously excluded class or method to the analysis without deleting the filter.
Exclude	You can tell JProbe to ignore code in a specified package, class, or method. The resulting snapshot contains no data on excluded classes and methods.
Native	By default, native (JNI) methods are excluded from the Coverage analysis. Select this action for specific native methods that you want to include. **Note** Avoid assigning this action to non-native methods or to packages or classes that contain non-native methods. If you do, the resulting data may be incorrect.

Triggers for Coverage

The **Analysis Type** > **Automation** tab allows you to set triggers and data recording for your analysis.

Triggers

Triggers provide a way to tie JProbe actions to your program's execution. You can have JProbe perform an action when your program enters or exits a method of your choice. With advanced trigger controls, you can also specify when to start executing the trigger actions and how long to execute them. Triggers are particularly useful for automated sessions.



Triggers are defined in the same way for all the analysis tools, although the trigger actions are somewhat different. This section describes the trigger actions for a Coverage analysis. To learn how to create triggers, see "Setting Triggers" on page 425.

The table below describes the trigger actions for Coverage.

Trigger Action	Description
Recording	Starts recording data. In the Initial Recording at JVM Start section, you can select the following options: • Data Recording Off • Data Recording On **Note** When the trigger action is set to *Recording*, the selected recording parameter is also displayed in the triggers table.

Trigger Action	Description
Snapshot	Takes a snapshot. The snapshot file name takes the form: *snapshot_<n>_raw.jpc*.

Recording Controls

You can use the data recording controls to start or stop recording data for a session by setting them in the **Automation** tab. The default setting is Data Recording On.

During a session, you can start or stop recording data using the toolbar buttons. If you want to use the toolbar button to control when data recording starts, set the Initial Recording at JVM Start option to Data Recording Off. For details, see “Controlling a Session with Runtime Toolbar Buttons” on page 374.

18

Exploring Coverage Analysis Views

This chapter provides detailed information about the Coverage views used to present coverage data.

This chapter contains the following sections:

Exploring the Coverage Session 374
Exploring the Snapshot Browser 378
Exploring the Coverage Source View 384
Setting Preferences for Coverage Views 390
Learning More About Coverage Results 393

Exploring the Coverage Session

The Coverage Session provides a real-time view of test coverage for your application as your session is running. The Coverage Session remains available when the session ends; you can toggle between it and other views.

This section contains the following information:

- Controlling a Session with Runtime Toolbar Buttons
- Understanding the Data in the Coverage Session
- Interacting with the Coverage Session

Controlling a Session with Runtime Toolbar Buttons

Data recording is set when you create a settings file, on the **Analysis Type** > **Automation** tab. The default setting is Data Recording On, which means that JProbe starts to record data as soon as your application's JVM starts. If you choose Data Recording Off, you can use the toolbar buttons in the Coverage Session to control when JProbe collects data. For example, you may want to get past all the startup activities in your application before beginning to collect data. In this case, you click the Start Recording toolbar button just before you begin your test case. For more information, see "Recording Controls" on page 372.

In addition to starting and finishing data collection for the test case, you can take a snapshot of data at any time during the session.

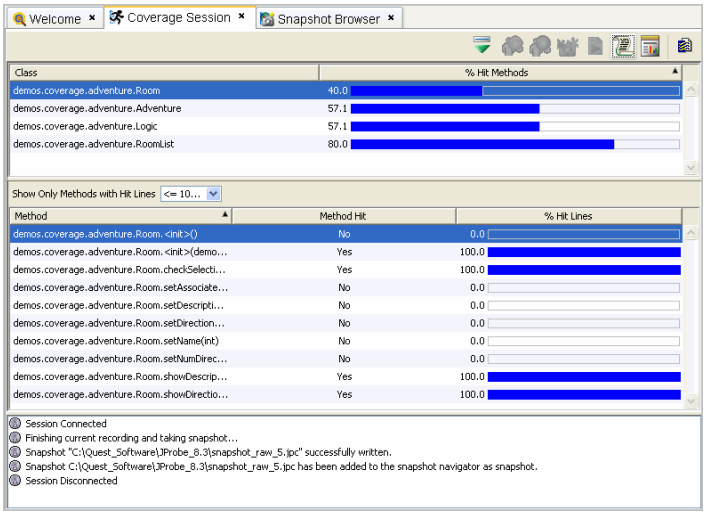
The following toolbar buttons are available during the session:

Icon	Action	Description
	Start Recording Data	Starts recording Coverage data.
	Stop Recording Data	Stops recording Coverage data and takes a snapshot.
	Take a Snapshot	Takes a snapshot.

Understanding the Data in the Coverage Session

Data in the Coverage Session is presented in terms of hits; that is, the classes, methods, and lines of code that were exercised by your test case. The Coverage Session has two tables. The upper table displays a list of classes hit by the test case and shows the percentage of methods that were hit within each class. When you click a class, the lower table shows the names of the methods in the class, whether or not the method was hit, and the percentage of lines that were hit.

Note Data about condition coverage (collected when Line with Condition is chosen in the configuration) is not included in the Coverage Session view. Condition data is saved to the snapshots taken during the session.



You can filter the methods displayed in the lower table using the *Show Only Methods with Hit Lines* field.

Note The *Show Only Methods with Hit Lines* field is enabled only if you have selected *Line* or *Line with Condition* as the coverage level option, when defining the Coverage analysis configuration.

This option filters the results by percentage of hit lines; only methods with a percentage of hits below the chosen percentage are displayed. For example, if you

choose <70%, the methods with 0% to 69% hit lines are displayed, while the methods with 70% to 100% coverage are hidden.

Tip You can change the color used for hit lines. For more information, see "Customizing the Colors Used to Display Results" on page 392.

Interacting with the Coverage Session

You can perform the following tasks:

Task	Action	Response
Examine the methods in a class	Choose the class in the upper table.	The lower table displays a list of methods in the chosen class.
Show only the methods with a percentage of hits less than a specified threshold	Set **Show Only Methods with Hit Lines** to a percentage of hit lines.	Methods with % Hit Lines *less than* the chosen percentage are shown. The rest of the methods are hidden.
Sort a table by a metric	Click the column heading for the metric.	The table is sorted. To reverse the sort order, click the heading again.
Show/hide columns	Right-click and choose **Show/Hide Columns**. Choose the columns to show and click **OK**.	The chosen columns are displayed. The rest are hidden.
Rearrange columns	Drag a column by its column heading and drop it in a new location.	The column appears in its new location.

Task	Action	Response
Resize column widths	• To set the width, drag the edge of the column heading left or right. • To size a column to fit its data, right-click the column and choose **Size <column> to Fit**. • To size all columns, right-click and choose **Size All to Fit**.	The affected columns are resized. When sizing all columns to fit, if the resized columns do not fill the table, the width of all columns is increased to fill the empty space.
Export table data	Right-click the table and select **Export to CSV**. See "Exporting Table Data to a CSV File" on page 442.	The file contains the table data separated by a comma. It may also contain headers.
View the execution log from the Runtime Summary view	Click **Runtime Execution Log**  .	The **Execution Log** dialog box opens and you can review the information by navigating through the available tabs.
View the history of events since the Console connected to the session.	Click **View Session History**  .	The **Session History** dialog box opens. In the left pane, you can parse the list and review the events that happened since the Console last connected to the session. In the right pane you can view additional details for select events.
Create a report for this view	Click **Generate Report**  . See "Generating HTML Reports" on page 440.	The report contains a summary of the classes and (optionally) a summary of the methods by class.

Task	Action	Response
Create a support bundle	Click **Create Support Bundle**  to automatically collect basic support information into a ZIP file, for submission to Quest Support.	The **Support Bundle Created** dialog box appears, indicating the location where the ZIP file was saved. You can click **Copy Bundle Name** to copy to the clipboard the name of the bundle that was just created.

Exploring the Snapshot Browser

The Snapshot Browser displays data from the currently chosen snapshot.

This section contains the following information:

- Opening the Snapshot Browser
- Understanding the Data in the Snapshot Browser
- Interacting with the Snapshot Browser

Opening the Snapshot Browser

The Snapshot Browser opens automatically when your application exits normally and displays the contents of the last snapshot taken. You can view other snapshots in the Snapshot Browser.

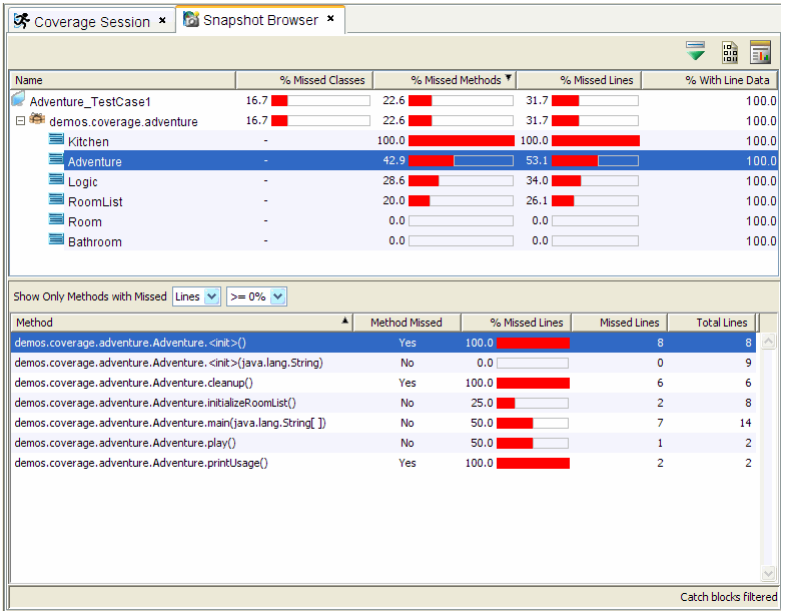
To open a different snapshot:

1 In the Snapshots Navigator, double-click the snapshot that you want to display.

Understanding the Data in the Snapshot Browser

Data in the Snapshot Browser is presented in terms of misses; that is, the classes, methods, lines of code, and conditions (if the snapshot contains condition coverage data) that were missed by your test case. The Snapshot Browser is divided into two areas. In the upper area, called the Application Tree, your application’s packages and

classes are organized in a tree format. When you choose a package or class in the tree, the Method Detail table displays data for each of its methods.


Coverage Session
Snapshot Browser
Name
% Missed Classes
% Missed Methods
% Missed Lines
% With Line Data
Adventure_TestCase1 16.7 22.6 31.7 100.0
demos.coverage.adventure 16.7 22.6 31.7 100.0
Kitchen - 100.0 100.0 100.0
Adventure - 42.9 53.1 100.0
Logic - 28.6 34.0 100.0
RoomList - 20.0 26.1 100.0
Room - 0.0 0.0 100.0
Bathroom - 0.0 0.0 100.0
Show Only Methods with Missed Lines >= 0%
Method
Method Missed
% Missed Lines
Missed Lines
Total Lines
demos.coverage.adventure.Adventure.<init>() Yes 100.0 8 8
demos.coverage.adventure.Adventure.<init>(java.lang.String) No 0.0 0 9
demos.coverage.adventure.Adventure.cleanup() Yes 100.0 6 6
demos.coverage.adventure.Adventure.initializeRoomList() No 25.0 2 8
demos.coverage.adventure.Adventure.main(java.lang.String[ ]) No 50.0 7 14
demos.coverage.adventure.Adventure.play() No 50.0 1 2
demos.coverage.adventure.Adventure.printUsage() Yes 100.0 2 2
Catch blocks filtered

You can filter the methods displayed in the lower table by using the Show Only Methods with Missed fields. You can filter by the percentage of missed lines of code or by the percentage of missed conditions; only methods that exceed the percentage are displayed. For example, if you choose Lines >50%, the only methods displayed are the ones where over half of the lines of code in the method were missed by the test case.

A Closer Look at Data in the Application Tree

The default view of the data shows a selection of percentage missed values for the application, packages, and classes listed in the Application Tree. The following table describes the values in the tree. The columns marked with an asterisk are shown by default. The other columns can be added to the view by right-clicking in the table to open the Show/Hide Columns dialog box, where you can choose other columns to be displayed.

Column Name	Description
% Missed Classes*	= Missed Classes / Total Classes
Missed Classes	If the snapshot contains baseline snapshot data, this column states the number of classes that were missed. Otherwise, this column will be zero.
Total Classes	If the snapshot contains baseline snapshot data, you see the total number of classes in your application (taking filters into account). Otherwise, you see the total number of classes with methods that were called by your test case (taking filters into account).
% Missed Methods*	= Missed Methods / Total Methods
Missed Methods	Number of methods missed by the test case.
Total Methods	Total number of methods in the class/package/application (taking filters into account).
% Missed Lines*	= Missed Lines / Total Lines
Missed Lines	Number of lines of code missed by the test case.
Total Lines	Total lines of code in the class/package/application (taking filters into account).
% With Line Data*	Percentage of lines of code that have line data.
% Missed Conditions*	= Missed Conditions / Total Conditions
Missed Conditions	Number of condition outcomes missed by the test case.
Total Conditions	Total number of possible condition outcomes in the class/package/application (taking filters into account).
% With Condition Data*	Percentage of lines of code with condition data.

For the Missed and Total columns, data is rolled up the tree. Values for the package represent the sum of all classes in the package, while the values attributed to the application represent the sum of all its packages.

The name of each field in the Application Tree is prefixed by an icon which provides information about that field, as described in the following table.

Icon	Type
	Application node.
	Package node.
	Class node.

A Closer Look at Data in the Method Detail Table

By default, each method states whether or not it was missed, the total lines of code in the methods, the total number of conditions (if available), and the name of the class. You can choose to show additional columns.

The following table describes the columns in the method detail table. The columns marked with an asterisk are shown by default. The other columns can be added to the view by right-clicking in the table to open the Show/Hide Columns dialog box, where you can choose other columns to be displayed.

Column Name	Description
Method*	Name of the method.
Method Missed*	States whether or not the method was missed.
% Missed Lines	= Missed Lines / Total Lines
Missed Lines	Number of lines in the method that were missed by your test case.
Total Lines*	Total number of lines of code in the method.

Column Name	Description
% Missed Conditions	= Missed Conditions / Total Conditions
Missed Conditions	Number of condition outcomes in the method that were missed by your test case.
Total Conditions*	Total number of possible condition outcomes in the method.

Interacting with the Snapshot Browser

You can perform the following tasks:

Task	Action	Response
Examine methods in the application	Choose the root of the tree.	The lower table displays all methods in the application.
Examine methods in a package	Choose a package branch in the tree.	The lower table displays all methods in the chosen package.
Examine methods in a class	Choose a class node in the tree.	The lower table displays all methods in the chosen class.
Show only the methods with a percentage of missed lines greater than a specified threshold	Set the **Show Only Methods with Missed** fields to **Line** and to a percentage of missed lines.	Methods with % Missed Lines *greater than* the chosen percentage are shown. The rest of the methods are hidden.
Show only the methods with a percentage of missed conditions greater than a specified threshold (requires condition data)	Set the **Show Only Methods with Missed** fields to **Conditions** and to a percentage of missed conditions.	Methods with % Missed Conditions *greater than* the chosen percentage are shown. The rest of the methods are hidden.

Task	Action	Response
View the source code for a class or method	Choose the class or method, right-click and choose **View Source**. You may be prompted to locate the source code file.	The source code is displayed in a source view. See "Exploring the Coverage Source View" on page 384.
Sort a table by a metric	Click the column heading for the metric.	The table is sorted. To reverse the sort order, click the heading again.
Show/hide columns	Right-click and choose **Show/Hide Columns**. Choose the columns to show and click **OK**.	The chosen columns are displayed. The rest are hidden.
Rearrange columns	Drag a column by its column heading and drop it in a new location.	The column appears in its new location.
Resize column widths	• To set the width, drag the edge of the column heading left or right. • To size a column to fit its data, right-click the column and choose **Size <column> to Fit**. • To size all columns, right-click and choose **Size All to Fit**.	The affected columns are resized. When sizing all columns to fit, if the resized columns do not fill the table, the width of all columns is increased to fill the empty space.
Export table data	Right-click the table and choose **Export to CSV**. See "Exporting Table Data to a CSV File" on page 442.	The file contains the table data separated by a comma. It may also contain headers.

Task	Action	Response
Create a report for this view	Click **Export Data to HTML**. See "Generating HTML Reports" on page 440.	The report contains the level of detail chosen in the Create Coverage Report dialog box.

Exploring the Coverage Source View

The Source view displays the source code for a chosen method.

This section contains the following information:

- Opening the Source Code
- Understanding the Data in a Source View
- Interacting with the Source View

Note You need access to the application's source code to use this view.

Opening the Source Code

You can have multiple Source views open. Each view is labeled with the name of the source code file displayed within it.

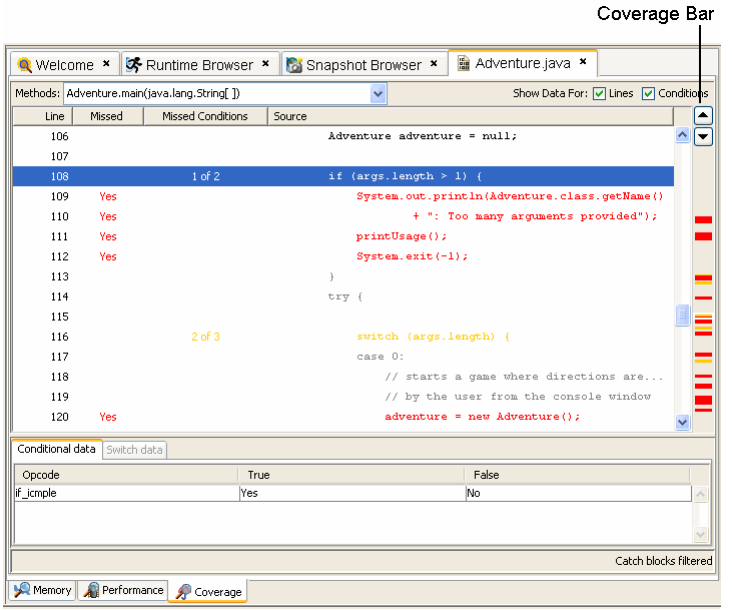
To view source code:

1. From the Snapshot Browser, do one of the following:
   - Double-click a package, class, or method.
   - Select a package, class, or method, and click the **View Source** toolbar button.
2. If prompted, locate the source code file for the class.

   The source code is displayed in a Source view.

Understanding the Data in a Source View

In a Source view, your code is color-coded to highlight missed lines of code and missed conditions (if collected). If you collected data at the Line with Condition level, and the snapshot contains data on conditions, you can choose a condition and see details about it in the Condition Details table.



A Closer Look at the Data in the Source Code

By default, the source code is color-coded to highlight missed lines of code and, if available, missed conditions:

- Missed lines, including condition statements with all outcomes missed, are **red**.
- Condition statements with some missed outcomes are **gold**.
- Code that was hit by your test case is **black**.
- Code for which there is no data is **gray**.
- Filtered out catch blocks are **gray**.

You can change these colors. For more information, see “Customizing the Colors Used to Display Results” on page 392.

Coverage Bar

Along the right margin is a navigation aid called the Coverage Bar. The length of the Coverage Bar represents all the source code in the file, not just the code currently displayed in the view. The colored stripes on the bar represent the relative size and position of missed lines of code and missed conditions (if available). You can click a stripe to navigate to that block of code. For more information, see “Interacting with the Source View” on page 388.

Missed Condition Column

When the snapshot contains data on conditions, the Missed Conditions column displays the number of missed outcomes within the context of the total number of possible outcomes. For example, if one out of two conditions were missed, the value “1 of 2” is displayed.

To understand how JProbe determines the total number of conditions, consider the following `if` statement:

```
if(a && b && c)
```

JProbe calculates two possible outcomes (true and false) for each of the three subconditions for a total of six possible outcomes (2+2+2=6). Note that the calculation is *not* based on the combinations that these possible outcomes can generate (2x2x2 = 8).

If the condition outcomes do not match your expectations, your compiler may have made some optimizations or rearrangements when creating bytecode. For more information, see “Nonintuitive Results for Conditions” on page 396.

Missed Column

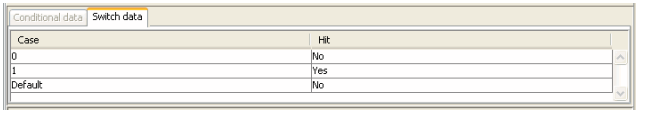
This column simply states if the line of code was missed.

A Closer Look at the Data in the Condition Detail Table

When you choose a condition in the source code, the Condition Detail table is populated with details about the condition coverage.

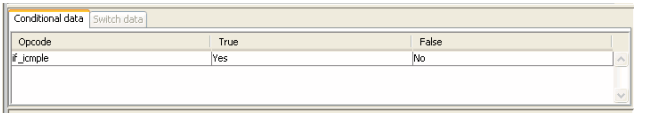
Switch data

If the condition is a `switch...case`, the table includes the possible case values including the implicit or explicit default, and whether or not each was executed.



Conditional data

For non-`switch` conditions, the table displays the conditional opcode mnemonics from the line, and for each indicates whether execution of the condition resulted in a true or false outcome.



If you are unfamiliar with opcodes, you may want to read the following background material. Your java compiler (for example `javac`) compiles Java source code into instruction opcodes known as bytecode for the Java virtual machine. Conditions in the source code are translated into condition opcodes in the JVM instruction set. The condition opcodes are like a simple `if` statement that only ever checks a single true/false condition. A compound condition in Java source such as `if (a && b)` becomes multiple condition opcodes in the compiled bytecode.

For more information on how compilers generate bytecode for the JVM, refer to *The Java Virtual Machine Specification* published by Sun Microsystems. In particular, see the following chapters:

- Chapter 6, “The Java Virtual Machine Instruction Set”
  http://java.sun.com/docs/books/jvms/second_edition/html/Instructions.doc.html
- Chapter 7, “Compiling for the Java Virtual Machine”
  http://java.sun.com/docs/books/jvms/second_edition/html/Compiling.doc.html

JProbe Coverage uses condition opcodes to collect condition coverage data. For each condition opcode, as your program runs JProbe will track which of the two possible outcomes (true or false) have occurred and record this in the snapshot. In the Source View of the snapshot, for each line which has a conditional opcode, the Conditional Data table displays the conditional opcode mnemonics from the line, and for each indicates whether execution of the condition resulted in a true or false outcome.

Note Opcodes generated by a compiler often differ from the source code. You may see a condition in your source code that does not evaluate as you expect. For more information, see “Nonintuitive Results for Conditions” on page 396.

The following table lists the opcodes that may appear in the Condition Detail table:

Opcode	Opcode Mnemonic	Stack Arguments	Description
153	ifeq	value	true if equal zero
154	ifne	value	true if not equal zero
155	iflt	value	true if less than zero
156	ifge	value	true if greater than or equal to zero
157	ifgt	value	true if greater than zero
158	ifle	value	true if less than or equal to zero
159	if_icmpeq	value1, value2	true if int equal
160	if_icmpne	value1, value2	true if int not equal
161	if_icmplt	value1, value2	true if int less than
162	if_icmpge	value1, value2	true if int greater than or equal
163	if_icmpgt	value1, value2	true if int greater than
164	if_icmple	value1, value2	true if int less than or equal
165	if_acmpeq	value1, value2	true if reference equal
166	if_acmpne	value1, value2	true if reference not equal
198	ifnull	value	true if reference is null
199	ifnonnull	value	true if reference is non null

Interacting with the Source View

You can perform the following tasks:

Task	Action	Response
Navigate to a block of missed code	Click a stripe in the Coverage Bar.	The block of missed code is displayed.
Navigate to the code represented by the next stripe in the Coverage Bar	Click the down arrow on the Coverage Bar or right-click and choose **Next Stripe**.	The next block of code (missed lines or missed conditions) is displayed.
Navigate to the code represented by the previous stripe in the Coverage Bar	Click the up arrow on the Coverage Bar or right-click and choose **Previous Stripe**.	The previous block of code (missed lines or missed conditions) is displayed.
Navigate to a method within the code	Choose a method from the Method drop-down list at the top of the Source view.	The method is displayed.
Examine a missed condition in more detail (requires conditions data)	Choose a missed condition in the source code.	Information about the chosen condition is displayed in the Condition Detail table.
Color-code source by missed lines and/or missed conditions (requires conditions data)	Beside Show Data For, choose **Lines**, **Conditions**, or both checkboxes.	The source code colors are adjusted accordingly.
Show/hide columns	Right-click and choose **Show/Hide Columns**. Choose the columns to show or hide and click **OK**.	The chosen columns are displayed. The rest are hidden.
Rearrange columns	Drag a column by its column heading and drop it in a new location.	The column appears in its new location.

Task	Action	Response
Resize column widths	• To set the width, drag the edge of the column heading left or right. • To size a column to fit its data, right-click the column and choose **Size <column> to Fit**. • To size all columns, right-click and choose **Size All to Fit**.	The affected columns are resized. When sizing all columns to fit, if the resized columns do not fill the table, the width of all columns is increased to fill the empty space.

Setting Preferences for Coverage Views

You set preferences in the Options dialog box. For more information, see “Setting Preferences” on page 453.

This section contains the following information:

- Displaying Percentages in Bar Charts
- Filtering Out Catch Block Data
- Customizing the Colors Used to Display Results

Displaying Percentages in Bar Charts

By default, JProbe displays percentage-based metrics in the context of a bar chart. You can hide the bar charts.

To hide bar charts:

1. Click the  **Options** toolbar button or choose **Tools > Options**.
2. Choose **Data Display > Coverage**.
3. To hide the bar charts, clear the **Show Bar Chart for Percentage Metrics** check box.
4. Click **OK**.

   When you return to a Coverage view, the bar charts are hidden.

Filtering Out Catch Block Data

The data for catch blocks is included in the following metrics:

- % Missed Lines
- Missed Lines
- Total Lines
- % Missed Conditions
- Missed Conditions
- Total Conditions

Catch blocks, however, are often impossible to test. If you choose not to test them, you may find it easier to interpret your results without the catch block data inflating the missed values. You can set a filter to remove the data from the calculations.

To filter out catch block data from calculations:

1. Click the  **Options** toolbar button or choose **Tools > Options**.
2. Choose **Data Display > Coverage**.
3. Choose **Filter out Catch Blocks**.
4. Click **OK**.

   When you return to a Coverage view, the calculations reflect the filter setting.

Exceptions

Although it is not generally considered good coding practice, it is possible that a line can have both normal statements and catch block statements.

Catch block filters do not apply in the followings situations:

- **Try and catch appear in the same line of code**

  ```
  try { Thread.sleep(1000); } catch (Exception e) {}
  ```

  If `Thread.sleep(1000)` is executed, coverage is 100%; if not, coverage is 0%.

- **Executed code and catch appear in the same line of code**

  ```
  try {
      System.out.print(obj.toString());
      System.out.println(); } catch (Exception e) {
          System.exit(1); }
  ```

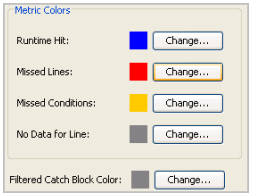
  If `System.out.print()` is executed, the coverage is 100% regardless of whether or not it throws an exception.

Customizing the Colors Used to Display Results

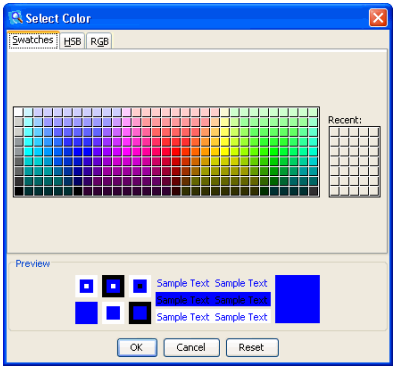
You can change the colors used to represent hit lines, missed lines, missed conditions, catch blocks that have been filtered out, and unexecuted code.

To set colors:

1. Click the  **Options** toolbar button or choose **Tools** > **Options**.
2. Choose **Data Display** > **Coverage**.



3. Click **Change** beside a metric's color.



4 Choose a new color and click **OK**.

5 Change other colors as desired.

6 Click **OK**.

When you return to a Coverage view, the new colors are used.

Learning More About Coverage Results

When reviewing results in the Coverage views, you may have some questions about how JProbe arrives at the data you see.

This section contains the following information:

- Nonintuitive Results for Some Source Code
- Nonintuitive Results for Conditions

For more information, see "Troubleshooting Coverage Analysis Results" on page 409.

Nonintuitive Results for Some Source Code

JProbe provides test case coverage results on executable source code. You may find instances, however, where JProbe appears to assign data to lines of source code that are clearly not executable, such as a comment or single brace. To understand how these results occur, you need to look to your `javac` compiler.

When your `javac` compiler compiles your source code into *.class* files, the class files contain executable bytecode. The compiler may also add debugging information that

maps the bytecode to lines of source code in the *.java* file. It saves this mapping in a line table in the class file. The mapping can sometimes be surprising, and it can vary with different compilers, compiler versions, `javac` options (eg. `-g`), or even the formatting of the source code. For example, `javac` may attribute bytecode to the closing brace (}) of a `void` method even if the method has no `return` statement, or may attribute it to the last line of code.

JProbe uses the mappings in the line table to match the coverage results for your bytecode to the lines in your source code. JProbe cannot modify these mappings.

The following table summarizes some of the nonintuitive results that you may see in JProbe and provides an explanation of the compiler behavior that causes these results:

Nonintuitive JProbe Results	Explanation of Compiler Behavior
A `class` declaration or its opening brace is reported as missed rather than unexecutable.	This happens when the compiler generates a default constructor for a class with no explicit constructor, but the constructor is not used. Most compilers generate a line table for this method that attributes the code to the `class` declaration itself–either the line with the `class` keyword or the opening brace (`{`). **Workaround:** To ignore the unused method in subsequent JProbe sessions, add a data collection filter: `com.example.MyClass.<init>():exclude`
Data is assigned inconsistently to the closing braces of blocks.	Most compilers inconsistently attribute code to the closing brace (`}`) of a block. In addition, sometimes the compiler generates code at the end of a block to skip subsequent blocks, such as subsequent `catch` blocks or the `else` portion of an `if` statement.

Nonintuitive JProbe Results	Explanation of Compiler Behavior
Data is assigned inconsistently to the closing braces of methods.	In bytecode, methods must always have an explicit `return` statement. In the Java language, however, the `return` statement is optional for `void` methods. For most compilers, if a `return` statement exists, the bytecode is assigned to it. Otherwise, the bytecode is assigned to the closing brace.
A `catch` line with no source code is reported as missed rather than unexecutable.	Most compilers generate code to save caught exceptions.This code is usually (but not always) attributed to the `catch()` clause itself rather than the block associated with it, whether or not that block is empty. Many test cases do not exercise this generated code, so JProbe reports it as missed. **Workaround:** To remove all data associated with catch blocks (including additional bytecode generated by the compiler), choose the "Filter out catch blocks" display option. For more information, see "Filtering Out Catch Block Data" on page 391.
The start or end of some `synchronized` blocks are unexpectedly colored as catch blocks when the JProbe "Filter out catch blocks" display option is enabled.	The bytecode to acquire and release the monitor associated with a `synchronized` block is normally attributed to the start and end of the block respectively. There is also, however, an implicit exception handler (catch block) generated to release the monitor in the event of an exception. The exception handler bytecode may be attributed to either the start or end of the block. JProbe identifies the affected lines in the `synchronized` block as a catch block.

Nonintuitive JProbe Results	Explanation of Compiler Behavior

Data is assigned inconsistently to one or more of the lines that make up a statement.	When source code for a statement is split over multiple lines, the compiler may attribute the bytecode for the statement to one or more of the lines of source code. For example, consider the following two statements:
	`for (i = 0;`
	`  someLongCondition(i);`
	`  ++i) { ...`
	`String str = prefix +`
	`  "." + suffix;`
	The compiler may attribute code to all three lines in the `for` statement, but only to the first line in the `String` concatenation.

Uninitialized variable declarations are reported as missed rather than unexecutable.	When creating the line table for a method, a few compilers assume that the start of the bytecode for the method can be attributed to the first line. If this first line is a statement with no executable code such as an uninitialized variable declaration (`int i;`), then no bytecode is created for the line. **Workaround:** • Initialize variable declarations (`int i = 0;`) • Move the declaration below the first line of executable code • Try another compiler
The last line of a `catch` block before a `finally` block is reported as hit, even though the rest of the `catch` block may be missed.	Some older compilers may misattribute code from a `finally` block to the last line of code in a preceding `catch` block.

Nonintuitive Results for Conditions

Compilation of Java source to bytecode may not always be as one might expect. Compilers may rearrange conditions, omit ones that are unnecessary, or add ones not obvious in the source, subject to some constraints by the Java language. They often

invert conditions so that a condition you expected to be reported as having been found true but not false, may be reported by JProbe as false but not true.

For example, the following source code:

```
if (s == null) {
   a();
} else {
   b();
}
```

may result in bytecode similar to the following:

```
    aload s
    ifnonnull skip
    invokevirtual a()
    goto end
skip:
    invokevirtual b()
end:
```

So while the source is true if `s` is null, the `ifnonnull` opcode is true if `s` is *not* null. JProbe cannot change how the compiler treats conditions. For more information, see "A Closer Look at the Data in the Condition Detail Table" on page 386.

The following table lists some other common nonintuitive results where conditions are added or completely removed:

Source Code	Explanation of Compiler Behavior
`assert`	Most compilers add one hidden condition to the line for the `assert`, which in turn adds two more possible outcomes to the total outcomes on the line.
`class`	The source construct `<classname>.class` is sometimes compiled into code that calls `Class.forName(<classname>)`. To avoid repeated calls, the compiler generates code to check a hidden field to see if it has already saved the result of `forName()`. If it has not, it calls `forName()` and saves the result. Otherwise, it just uses the saved result. JProbe may include this check in its total possible condition outcomes.

Source Code	Explanation of Compiler Behavior

`debug`	Some compilers may remove debug conditions. For example, the following source code may not appear in the bytecode:
	`private static final boolean debug = false;`
	`...`
	`if(debug) System.out.println("Debug");`

19

Reporting Coverage Analysis Results

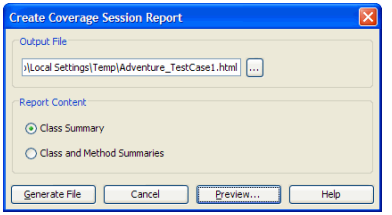
You can generate reports based on some of the Coverage views. This chapter presents the Coverage reports and the report options. To learn the steps to create a report, see “Generating HTML Reports” on page 440.

This chapter contains the following sections:

Coverage Session Report 400
Snapshot Browser Report 401

Coverage Session Report

All Coverage Session reports include a report header and a summary of the classes in the Coverage Session view. You can also choose to include a summary of the methods.

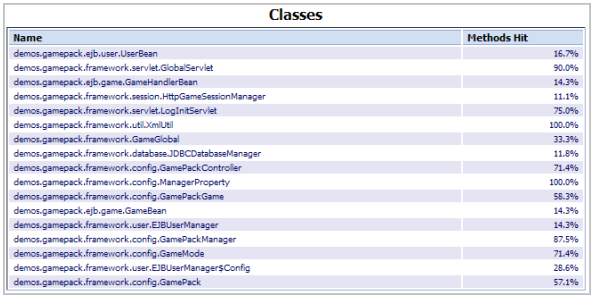


You can create the following reports:

- Class Summary
- Class and Method Summaries

Class Summary

The Class Summary contains a list of classes in the runtime view and the percentage of methods that were hit by the test suite.

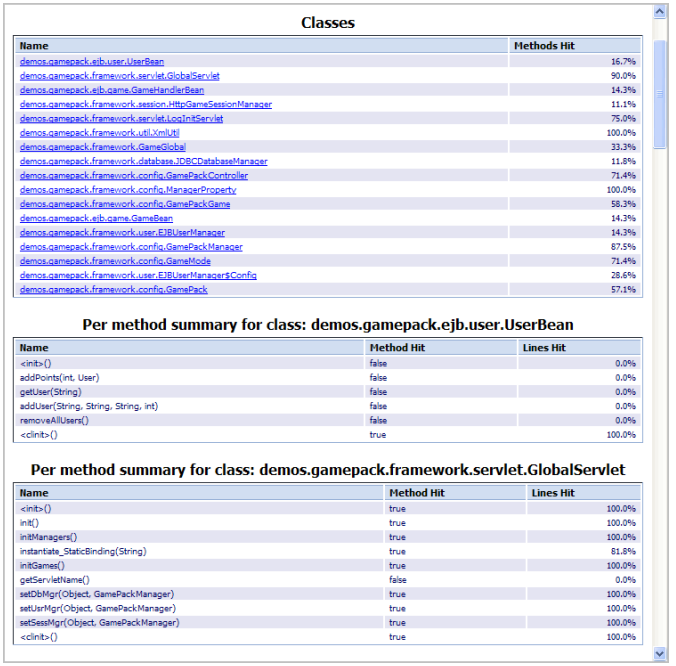


Classes

Name	Methods Hit
demos.gamepack.ejb.user.UserBean	16.7%
demos.gamepack.framework.servlet.GlobalServlet	90.0%
demos.gamepack.ejb.game.GameHandlerBean	14.3%
demos.gamepack.framework.session.HttpGameSessionManager	11.1%
demos.gamepack.framework.servlet.LogInitServlet	75.0%
demos.gamepack.framework.util.XmlUtil	100.0%
demos.gamepack.framework.GameGlobal	33.3%
demos.gamepack.framework.database.JDBCDatabaseManager	11.8%
demos.gamepack.framework.config.GamePackController	71.4%
demos.gamepack.framework.config.ManagerProperty	100.0%
demos.gamepack.framework.config.GamePackGame	58.3%
demos.gamepack.ejb.game.GameBean	14.3%
demos.gamepack.framework.user.EJBUserManager	14.3%
demos.gamepack.framework.config.GamePackManager	87.5%
demos.gamepack.framework.config.GameMode	71.4%
demos.gamepack.framework.user.EJBUserManager$Config	28.6%
demos.gamepack.framework.config.GamePack	57.1%

Class and Method Summaries

The Method Summary breaks down the Class Summary by class. For each method in a class, you see whether or not the method was hit by your test case and the lines of code (expressed as a percentage) that were hit.



Classes

Name	Methods Hit
demos.gamepack.ejb.user.UserBean	16.7%
demos.gamepack.framework.servlet.GlobalServlet	90.0%
demos.gamepack.ejb.game.GameHandlerBean	14.3%
demos.gamepack.framework.session.HttpGameSessionManager	11.1%
demos.gamepack.framework.servlet.LogInitServlet	75.0%
demos.gamepack.framework.util.XmlUtil	100.0%
demos.gamepack.framework.GameGlobal	33.3%
demos.gamepack.framework.database.JDBCDatabaseManager	11.8%
demos.gamepack.framework.config.GamePackController	71.4%
demos.gamepack.framework.config.ManagerProperty	100.0%
demos.gamepack.framework.config.GamePackGame	58.3%
demos.gamepack.ejb.game.GameBean	14.3%
demos.gamepack.framework.user.EJBUserManager	14.3%
demos.gamepack.framework.config.GamePackManager	87.5%
demos.gamepack.framework.config.GameMode	71.4%
demos.gamepack.framework.user.EJBUserManager$Config	28.6%
demos.gamepack.framework.config.GamePack	57.1%

Per method summary for class: demos.gamepack.ejb.user.UserBean

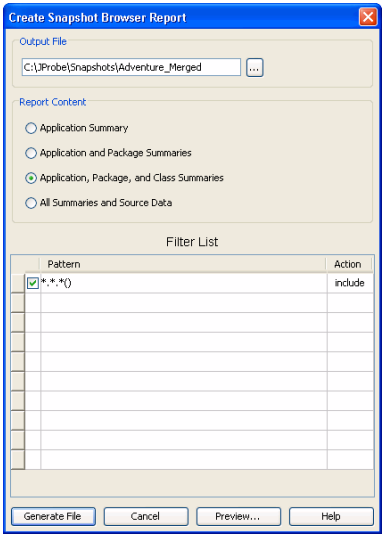
Name	Method Hit	Lines Hit
<init>()	false	0.0%
addPoints(int, User)	false	0.0%
getUser(String)	false	0.0%
addUser(String, String, String, int)	false	0.0%
removeAllUsers()	false	0.0%
<clinit>()	true	100.0%

Per method summary for class: demos.gamepack.framework.servlet.GlobalServlet

Name	Method Hit	Lines Hit
<init>()	true	100.0%
init()	true	100.0%
initManagers()	true	100.0%
instantiate_StaticBinding(String)	true	81.8%
initGames()	true	100.0%
getServletName()	false	0.0%
setDbMgr(Object, GamePackManager)	true	100.0%
setUsrMgr(Object, GamePackManager)	true	100.0%
setSessMgr(Object, GamePackManager)	true	100.0%
<clinit>()	true	100.0%

Snapshot Browser Report

The Snapshot Browser reports include a report header and a summary of the packages in the snapshot. You can choose to include progressively more detailed information about the packages, classes, and methods in the snapshot. By default, the report includes data on all code in all packages. You can choose to exclude some code from the report using report filters.

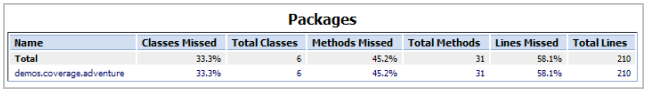


You can create reports that include the following information:

- Application Summary
- Package Summary
- Class Summary
- Source Data
- Report Filters

Application Summary

The Application Summary provides an overview of your application. For each package, it shows the total number of classes, methods, and lines of code and the percentage of each that were missed by your test suite. This summary is useful for communicating Coverage results to senior management.

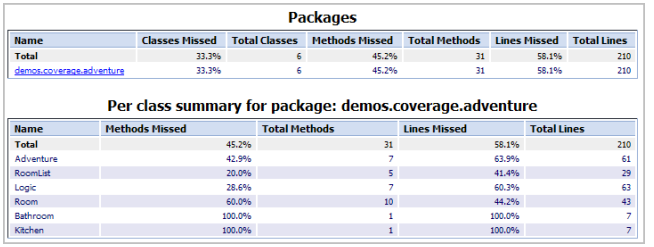


Packages

Name	Classes Missed	Total Classes	Methods Missed	Total Methods	Lines Missed	Total Lines
Total	33.3%	6	45.2%	31	58.1%	210
demos.coverage.adventure	33.3%	6	45.2%	31	58.1%	210

Package Summary

The Package Summary breaks down the results in the Application Summary by package. For each class in a package, you get the total number of methods and lines of code in the package and the percentage of missed methods and lines. You can use this summary to get an overview of test case coverage. This summary is also useful as a management tool; you can assign classes with a high percentage of missed code to members of your team for further investigation.



Packages

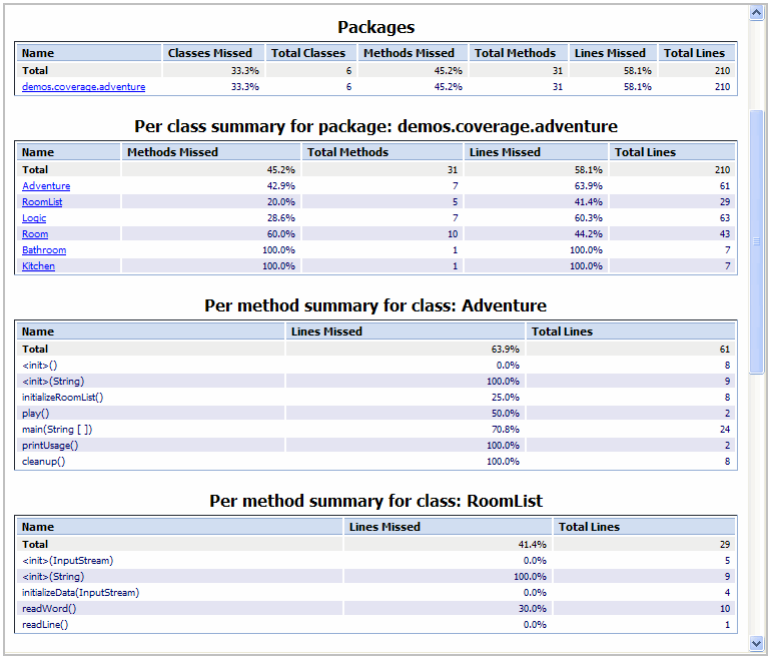
Name	Classes Missed	Total Classes	Methods Missed	Total Methods	Lines Missed	Total Lines
Total	33.3%	6	45.2%	31	58.1%	210
demos.coverage.adventure	33.3%	6	45.2%	31	58.1%	210

Per class summary for package: demos.coverage.adventure

Name	Methods Missed	Total Methods	Lines Missed	Total Lines
Total	45.2%	31	58.1%	210
Adventure	42.9%	7	63.9%	61
RoomList	20.0%	5	41.4%	29
Logic	28.6%	7	60.3%	63
Room	60.0%	10	44.2%	43
Bathroom	100.0%	1	100.0%	7
Kitchen	100.0%	1	100.0%	7

Class Summary

The Class Summary breaks down the results in the Package Summary by class. For each method in a class, you get the total number of lines of code in the method and the percentage of missed lines. A QA analyst might generate this report before investigating missed code and after modifying test suites. He can then use these reports to communicate before and after results to his manager.



Packages

Name	Classes Missed	Total Classes	Methods Missed	Total Methods	Lines Missed	Total Lines
Total	33.3%	6	45.2%	31	58.1%	210
demos.coverage.adventure	33.3%	6	45.2%	31	58.1%	210

Per class summary for package: demos.coverage.adventure

Name	Methods Missed	Total Methods	Lines Missed	Total Lines
Total	45.2%	31	58.1%	210
Adventure	42.9%	7	63.9%	61
RoomList	20.0%	5	41.4%	29
Logic	28.6%	7	60.3%	63
Room	60.0%	10	44.2%	43
Bathroom	100.0%	1	100.0%	7
Kitchen	100.0%	1	100.0%	7

Per method summary for class: Adventure

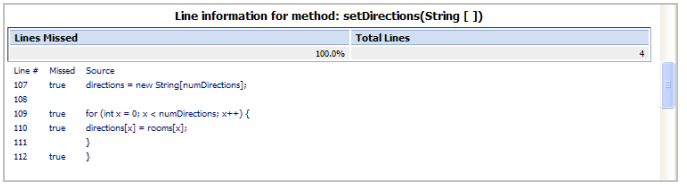
Name	Lines Missed	Total Lines
Total	63.9%	61
<init>()	0.0%	8
<init>(String)	100.0%	9
initializeRoomList()	25.0%	8
play()	50.0%	2
main(String [ ])	70.8%	24
printUsage()	100.0%	2
cleanup()	100.0%	8

Per method summary for class: RoomList

Name	Lines Missed	Total Lines
Total	41.4%	29
<init>(InputStream)	0.0%	5
<init>(String)	100.0%	9
initializeData(InputStream)	0.0%	4
readWord()	30.0%	10
readLine()	0.0%	1

Source Data

You can add the source code to your report. For each method, the source code is displayed along with line numbers and whether or not the line was missed. This report is useful to have with you if you want to talk to a developer about a particular method. You may also find it useful for analyzing Coverage results away from the computer.


Line information for method: setDirections(String [ ])

Lines Missed	Total Lines
100.0%	4

Line #	Missed	Source
107	true	directions = new String[numDirections];
108		
109	true	for (int x = 0; x < numDirections; x++) {
110	true	directions[x] = rooms[x];
111		}
112	true	}

Report Filters

By default, the report includes data on all code in all packages in the snapshot. You can choose to exclude some code from the report using report filters. Adding report filters is the same as adding application filters or data collection filters.

To create a list of report filters:

1. Click an empty cell and do one of the following:
   - To select a package or class, click the cell browse button, navigate to the package or class, and click **Open**.
   - To select a method, click the cell browse button and navigate to the class containing the method. When you select the class, you can expand its list of methods. Locate and double-click the desired method.
   - To select code from an element such as a WAR, EAR, or JAR file, click the cell browse button and navigate to the element. When you select the element, you can expand its contents. Locate and double-click the desired code.
   - Enter the package, class, or method (no signature) in the cell. For more information, see “Syntax for Filters” on page 406.

   The first cell in the row indicates whether or not the filter is enabled.
2. Set the action to **Include** or **Exclude** the code from the report.

   Tip You cannot interact with the Action cell until you specify a method.
3. Add additional filters. If two filters refer to the same code, the filter lower in the table is used.
4. You can change your list of filters.
   - To move a filter higher or lower in the table, drag the filter to its new location.
   - To disable a filter, deselect the checkbox.
   - To delete a filter, select the row, right-click, and select **Delete**.

Syntax for Filters

Filters are case-sensitive and blank spaces are not allowed. You can use an asterisk (*) as a wild card; it matches any character, including the package separator. If you specify the code in a shorthand or partial form, JProbe automatically changes it to canonical form. The change may not be exactly what you desire, so be sure to verify it. JProbe does not check to see whether the filter actually matches anything in your code. For example, if you make a typing error in your package name, nothing in your code will match the filter and so the filter is not effective.

The following table contains common syntax and describes how JProbe interprets it:

If you type this:	Canonical form is this:	And filters are applied to:
`*`	`*.*.*()`	All methods in all classes in all packages (including the unnamed package)
`.*`	`.*.*()`	All methods in all classes in the unnamed package only
`C`	`*.C.*()`	All methods in classes named `C` in any package
`.C`	`.C.*()`	All methods in the class named `C` in the unnamed package only
`M()`	`*.*.M()`	Method `M` in all classes in all packages
`P.String*`	`P.String*.*()`	All methods in any class in the package `P` whose name starts with `String` and all methods in any class in a package or subpackage of `P.String*` (matches both `P.StringArt.draw()` and `P.Stringent.Check.English.spelling()`)

If you type this:	Canonical form is this:	And filters are applied to:
`P.*.C.do*()`	*no change*	Any method whose name starts with `do` in any class named `C` in any subpackage of `P`, except `P` directly (matches `P.SubPkg.C.doGet()`, but does not match `P.C.doNotMatch()`)
`Pre*`	`*.Pre*.*()`	All methods in all classes starting with `Pre` in any package, as well as any class in a subpackage whose name starts with `Pre` (matches both `Pkg.Prepare.m()` and `Pkg.Pretend.C.m()`)
`s*y`	`*.s*y.*()`	All methods in classes whose name ends with `y` and either starts with `s` or belongs to a subpackage that starts with `s` (matches both `java.rmi.server.RMISocketFactory`, `com.quest.say`, and `java.security.cert.X509CRLEntry`)
`foo.bar`	`foo.bar.*()`	All methods in the class `bar` in package `foo`. This is likely not what you wanted. See the next example.
`foo.bar.*`	`foo.bar.*.*()`	All methods in all classes in `foo.bar` or any of its subpackages
`foo.bar.String.t*e()`	*no change*	Methods whose names start with `t` and end in `e` in `foo.bar.String` (matches both `toUpperCase()` and `toLowerCase()`)

20

Troubleshooting Coverage Analysis Results

As you work with JProbe, you may need help to get the most out of your analysis sessions. This chapter presents a few tips for solving some Coverage analysis problems and for running sessions efficiently. For Coverage analysis issues that are not covered here, refer to Quest's technical support.

This chapter contains the following sections:

Coverage Troubleshooting Tips.....410

Coverage Troubleshooting Tips

Lines of code with identical statements show different results

Identical source statements may be compiled differently by `javac`, or the line table generated by `javac` may have assigned the bytecode to source lines differently. For more information, see “Nonintuitive Results for Some Source Code” on page 393.

A statement that I think should be executable is non-executable (or vice versa)

Your `javac` compiler makes a determination about whether a line of code is executable or not. JProbe cannot change this assessment. You will need to become familiar with your compiler’s behavior. For more information, see “Nonintuitive Results for Some Source Code” on page 393.

When there are multiple statements on the same line, the line is reported as hit even if some statements were not executed

If a source line has any bytecode attributed to it by the line table generated by javac, JProbe considers the line executable.

Multiple statements will normally all contribute bytecode to the line, so execution of any of them causes JProbe to consider the line as hit.

For example:

```
if (isBad()) return; func()"
```

If `isBad()` is true, the statement calling `func()` will not be called, but the line will be considered hit. If `isBad()` is false, the `return` statement will not be executed, but the line will be considered hit.

The best way to resolve this (which is usually considered good programming practice anyway) is to put at most one statement per line:

```
if (isBad())
    return;
func();
```

The total number of lines of code reported by JProbe does not match the number of lines in my source code

When determining the total lines of code available, JProbe includes only code that is potentially executable, so blank lines and comments are ignored, as well as most lines

that contain only an opening or closing brace. Other factors can affect the total. For more information, see “Nonintuitive Results for Some Source Code” on page 393.

The condition results for a line are not what I expect.

The compiler can introduce conditions into the bytecode that do not explicitly exist in the source code. These hidden conditions affect the total number of possible outcomes. For more information, see “Nonintuitive Results for Conditions” on page 396.

An EJB class appears twice in my results

Some J2EE application servers, such as BEA WebLogic 8.1, may load EJBs twice using different classloaders. JProbe treats these instances as different EJB classes and tracks Coverage data separately.

21

Learning How to Use JProbe Effectively

This chapter provides background information on how JProbe works and suggests some things that you can do to use JProbe effectively.

This chapter contains the following sections:

Hints for Success with JProbe 414
Learning How JProbe Works 416
Demos and Tutorials 418
A Quick Tour of the Interface 419

Hints for Success with JProbe

The more familiar you are with the application that you want to analyze, the easier it will be for you to interpret the results of your JProbe analysis.

This section helps you prepare to use JProbe:

- Understanding the Architecture for Your Program
- Selecting Use Cases
- Compiling Your Program

Understanding the Architecture for Your Program

Review the architecture document for your program. Architecture documents typically outline a high-level view of the relationships within your Java program. For example, in a multitiered application, the architecture would identify what each tier does, how they communicate with each other, and what data they exchange. In addition, a core set of principal classes, methods, and interfaces should be identified. It is important that you understand the lifecycle of the principal objects—when they are created, how long they live, and when they can be removed. Focus on understanding the control flow, that is, how the objects within your program relate to one another.

If you do not have an architecture document, consider creating one. The Performance Summary and Methods views are a good place to start, as they can give you an overview of the control flow in your program.

Selecting Use Cases

Use cases describe high-level tasks from a user's perspective. If your development environment is process-conscious, look for documented use cases wherever project information is kept. In less formal development environments, you may need to derive the use cases by examining the program and end-user documentation. By focusing your efforts on frequently attempted use cases, you can ensure that your improvements will be visible to your users.

To help ensure that performance gains are visible:

1. Identify the use cases for your program.
2. Analyze which use cases are attempted most frequently, that is, determine the critical path of your program.

3. Identify the use cases in the critical path that are prompting complaints. You will focus on this subset of use cases first.
4. Formulate hypotheses for the use cases.
5. Run a Memory analysis on the use cases.
6. Run a Performance analysis on the use cases.
7. Run a Coverage analysis with the test cases that target the use cases.

For example, in a web-based application, a high-level task might be to log on to a secure area with a user ID and password. This case implies a set of steps that your user goes through, such as obtaining a unique ID, setting a password, entering the data into an online form, submitting the form for validation, and either going to the secure site or being denied access. If the validation step is a perceived problem, you could run Memory and Performance analysis sessions on this area only. With Coverage, you can ensure that the entire process is tested.

Compiling Your Program

You can run JProbe without recompiling your program for most situations. If you intend to analyze your program line-by-line, however, you may want to consider using the `-g` compiler option when you compile your program.

Improving Line-level Data with the -g Compiler Option

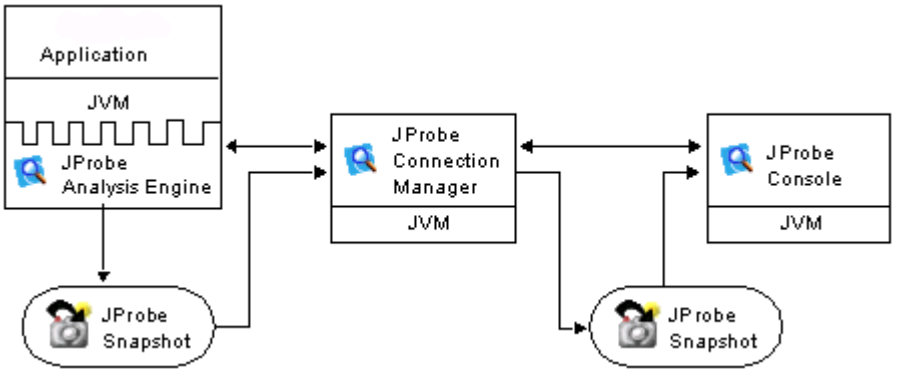
JProbe offers line-level analysis with its Performance analysis tool and its Coverage analysis tool. To report on line-level data, JProbe reads from the line number table created when you compiled your code. The line number table matches a range of bytes in the compiled bytecode to a line in your source code. Depending on your compiler, the match can sometimes be imprecise. You can view your existing line and local variables tables by running `javap -l` (letter “l”) with your compiled class file.

If you want to try to improve the quality of line information, you can compile your code with the `-g` compiler option. The `-g` option generates debugging information on line numbers and local variables, and identifies the source file names.

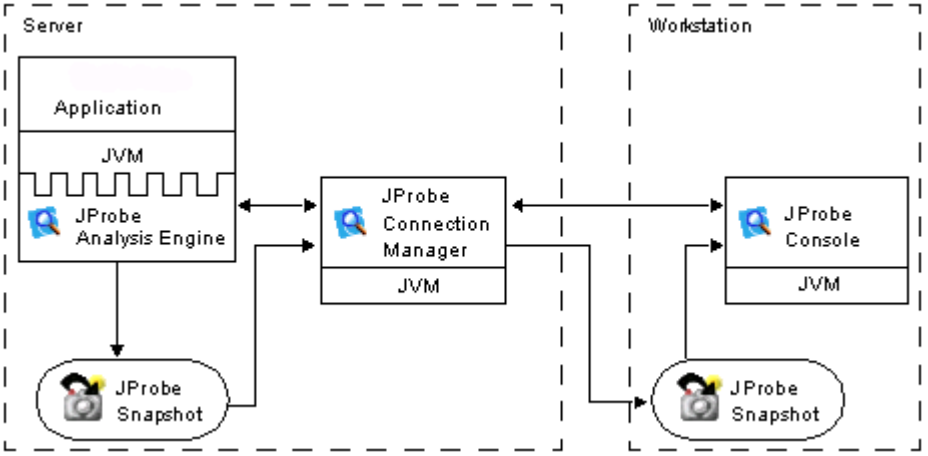
Learning How JProbe Works

JProbe has four main components:

- JProbe Analysis Engine – collects data on your application
- JProbe Connection Manager – manages communications
- JProbe Console – presents data in a graphical interface
- JProbe Snapshots – contains the data collected about your application



JProbe running on a single computer (local session)



JProbe running in a networked environment (remote session)

JProbe Analysis Engine

The JProbe Analysis Engine collects data on your application as it runs within its JVM. The selected JVM must support the current Sun JVM profiling interface, called JVMTI. The profiling interface provides the hooks that the JProbe Analysis Engine needs to analyze your application. The profiling interface may add some overhead, so do not expect your application to run as quickly as it would without the use of the profiling interface. Programs that execute very quickly are the most sensitive to this overhead.

JProbe Connection Manager

The JProbe Connection Manager runs in its own JVM and handles communications between the JProbe Analysis Engine and a JProbe Console. Only one JProbe Console can be connected to the JProbe Connection Manager at a time. The JProbe Connection Manager also transfers snapshots from the Analysis Engine to the Console.

JProbe Console

The JProbe Console provides you with a graphical interface where you can create a configuration, monitor your analysis session, and review snapshots of the analysis data. The JProbe Console runs in its own JVM, separate from the one running your

application. Why a separate virtual machine? Most importantly, you can be sure that the activity of the JProbe Console does not contribute to the data collected on your program. Secondly, on a system with multiple processors, you can take advantage of the extra processing power by running the JProbe Console on one processor and your program and the Analysis Engine on another. Lastly, with a dual license and remote connection, this architecture allows you to run your program on a server while monitoring it from a remote workstation.

JProbe Snapshots

Snapshots contain the data collected up to the time the snapshot is taken. Whether snapshots are taken manually by you, or automatically during or at the end of the session, they are saved as temporary files. In a remote session, the temporary snapshots are transferred to the local computer and removed from the remote computer. To keep snapshots, you need to rename them or move them to a different location.

Each analysis tool has its own file format for snapshots. You can distinguish among them by their icons and file extensions, as shown in the following table:

Icon	Analysis Tool	File Extension
	Memory	.jph
	Performance	.jpp
	Coverage	.jpc

Demos and Tutorials

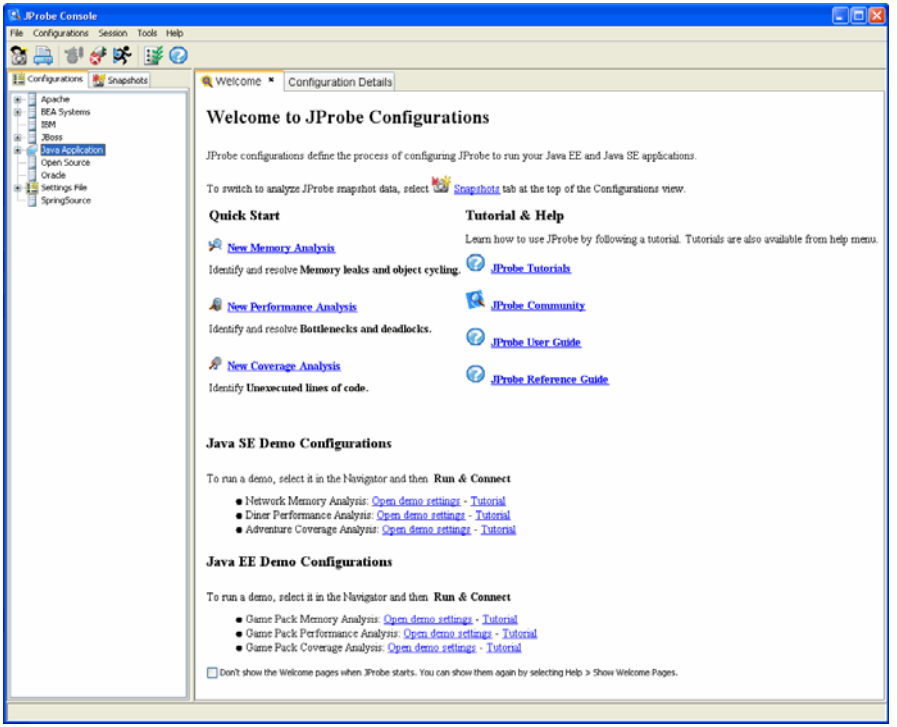
JProbe ships with a set of demo applications. They are located in the JProbe installation directory under *demos*. Each demo has an inherent problem that JProbe can identify. Some of the sample applications have tutorials associated with them. Others are explained within the main *.java* file for the sample application.

The tutorials are available in a PDF called *JProbe Tutorials*. If this PDF is not included in your installation, you can download it from the JProbe Community site.

A Quick Tour of the Interface

The JProbe Console has a main menu, a toolbar, a Navigation area (left pane), and a View area (right pane).

The Navigation area contains two tabs: **Configurations** and **Snapshots**. They are referred to as Configurations Navigator and Snapshots Navigator, respectively. The View area displays different information depending on the tab selected on the Navigation area.


JProbe Console
File Configurations Session Tools Help
Configurations
Snapshots
Apache
BEA Systems
IBM
JBoss
Java Application
Open Source
Oracle
Settings File
SpringSource
Welcome
Configuration Details
Welcome to JProbe Configurations
JProbe configurations define the process of configuring JProbe to run your Java EE and Java SE applications.
To switch to analyze JProbe snapshot data, select Snapshots tab at the top of the Configurations view.
Quick Start
New Memory Analysis
Identify and resolve Memory leaks and object cycling.
New Performance Analysis
Identify and resolve Bottlenecks and deadlocks.
New Coverage Analysis
Identify Unexecuted lines of code.
Tutorial & Help
Learn how to use JProbe by following a tutorial. Tutorials are also available from help menu.
JProbe Tutorials
JProbe Community
JProbe User Guide
JProbe Reference Guide
Java SE Demo Configurations
To run a demo, select it in the Navigator and then Run & Connect
Network Memory Analysis: Open demo settings - Tutorial
Diner Performance Analysis: Open demo settings - Tutorial
Adventure Coverage Analysis: Open demo settings - Tutorial
Java EE Demo Configurations
To run a demo, select it in the Navigator and then Run & Connect
Game Pack Memory Analysis: Open demo settings - Tutorial
Game Pack Performance Analysis: Open demo settings - Tutorial
Game Pack Coverage Analysis: Open demo settings - Tutorial
Don't show the Welcome pages when JProbe starts. You can show them again by selecting Help > Show Welcome Pages.

Configurations Navigator

The Configurations Navigator displays the list of configurations already defined for each of the Java applications and Java EE applications servers supported by JProbe. You can also define new configurations, modify existing ones, and load saved configuration files.

When the **Configurations** tab is selected on the Navigation area, the View area displays the **Welcome** to JProbe Configurations tab, which provides an overview of the JProbe configuration process. It also includes a quick way to launch the **New Configuration Wizard**, and allows you to define configuration settings for each JProbe analysis type. The tab also contains links to the JProbe community page, documentation, and tutorials created for several demo applications shipped with the product.

The View area also displays a **Configuration Details** tab, which guides you through the process of configuring JProbe to analyze a Java SE application or a Java EE application server that is currently not defined.

When you select a configuration in the Configurations Navigator, the View area displays a ***<Configuration_File_Name.jpl>*** tab. This tab allows you to edit the settings for the selected configuration; create a JPL file and a script file to integrate your application with JProbe; run your script file; or run your script file and auto-connect your application with JProbe.

Note JProbe demo configurations delivered with the product cannot be modified or deleted.

Snapshots Navigator

The Snapshots Navigator displays snapshots taken during analysis sessions. You can also load previously saved snapshots.

The View area by default displays the **Welcome** to JProbe tab. This tab contains three tabs across the bottom, one for each of the analysis types (**Memory**, **Performance**, and **Coverage**), and includes Quick Start links, shortcuts to JProbe community page and to documentation.

The View area displays the **Welcome** to JProbe tab until you connect to a JProbe session. When the JProbe Console connects, the current analysis tab updates to match the analysis type selected in configuration. It then displays the runtime view for that analysis type. You can interact with the runtime view (for example, you can update the runtime view to see the data collected so far, you can start and finish a use case, and take a snapshot).

Each analysis type can have one open snapshot. If a snapshot is open and you run another session under the same analysis type, the runtime view is for the current session, but any other open views continue to display data for the open snapshot. To display data from a different snapshot, double-click the snapshot entry in the Snapshots Navigator.

22

Refining Data Collection and Presentation

This chapter provides detailed information about JProbe filters, triggers, and categories.

This chapter contains the following sections:

Refining Data Collection Filters 422
Setting Triggers 425
Creating and Managing Categories 432

Refining Data Collection Filters

You can narrow your data collection by excluding some code from the analysis. You can also expand your data analysis to include other custom packages or third-party packages. Bear in mind that the more data you track, the slower the session runs and the more complicated the results will be.

This section contains the following information:

- Adding Filters
- Changing Filter Execution Order
- Removing Filters

Adding Filters

A filter is composed of two pieces of information: code and action. JProbe proposes default data collection filters based on how you described the code you are interested in (see the My Code screen of the Create/Edit Settings wizard). You can modify the data collection filters to increase or reduce the amount or complexity of data collected.

Caution Filter order is important. For more information, see "Changing Filter Execution Order" on page 424.

The procedure for adding filters is the same for all the analysis tools, however the filter actions differ among the tools.

Note You can add data collection filters during the configuration wizard on the **Specify JProbe Analysis** screen, or when editing the configuration via the **Analysis Type** tab.

To add a filter:

1. In the Runtime Summary view, click the **Set Recording Level** icon on the toolbar, then select **Record Data as Specified in the Table Below** option to activate the Data Collection Filters table.
2. Click an empty cell in the Data Collection Filters table and do one of the following:
   - To select a package or class, click the cell browse button, navigate to the package or class, and click **Open**.

- To select a method, click the cell browse button and navigate to the class containing the method. When you select the class, you can expand its list of methods. Locate and double-click the desired method.
- To select code from an element such as a WAR, EAR, or JAR file, click the cell browse button and navigate to the element. When you select the element, you can expand its contents. Locate and double-click the desired code.
- Enter the package, class, or method (no signature) in the cell. For more information, see "Syntax for Filters" on page 41.

The checkmark at the beginning of the row indicates that the filter is enabled.

Tip To enable/ disable all defined filters at once, you can select/ clear the check box beside the *Data Collection Filters* table title.

3 To set the data collection action that should occur for the specified code, click the Action cell and select an action. For more information about actions, see:

- "Filters for Memory" on page 124
- "Filters for Performance" on page 267
- "Filters for Coverage" on page 369

Tip You can interact with the Action cell only after you specify a method.

4 Click **Save**.

Syntax for Filters

Category filters use the same syntax as other JProbe application filters. For more information, see "Syntax for Filters" on page 41.

Using My Code Packages to Add Filters

You can add data collection filters to the current configuration based on the application filters that you specified in other configurations. Recall that application filters are set during the configuration wizard on the Specify Your Code screen.

To add filters from previously defined configurations:

1 Click **Add My Code**.

The list contains your configurations and the package-level application filters that you specified for each of them.

2 Choose packages and click **OK**.

The chosen packages are listed in the Data Collection Filters table. The filter action is set to the default action for the chosen analysis type. The new filters are enabled by default.

Changing Filter Execution Order

Filter order is important. When two entries refer to the same code, the entry lower in the table is used. A red line appears under filters when there is a conflict.

Task	Action	Response
Move a filter	Click the button at the left of the filter's row and drag the filter to the new location.	The filter is inserted where you drop it.
Insert an empty row between two filters	Right-click the button at the left of the filter's row and choose **Insert**.	The filter is inserted above the chosen filter row.

Removing Filters

You can remove filters in one of two ways. If you think you may use the filter again, you can temporarily disable the filter. The filter will not be used in this session. Otherwise, you can delete the filter.

Caution Order is important when using filters. Be careful when disabling filters that you do not inadvertently remove something that you want to analyze. For more information, see "Changing Filter Execution Order" on page 424.

Task	Action	Response
Disable a filter	Clear the check box beside the filter's row.	The filter is ignored during the subsequent analysis.
Re-enable a filter	Select the check box beside the filter's row.	The filter is applied during the subsequent analysis.

Task	Action	Response
Delete a filter	Right-click the button at the left of the filter's row and select **Delete**.	The row is removed from the table.

Setting Triggers

You can use triggers to perform actions automatically on entry to or exit from a method. Actions include recording data, taking a snapshot, and requesting a garbage collection. For example, you can wait to start collecting data until your application calls a specific method. You can further refine triggers by specifying when and how many times to execute the trigger based on either the number of method invocations or elapsed time.

Tip Triggers are required for most automated sessions. If you are interested in running automated sessions, see the *JProbe Reference Guide*.

This section contains the following information:

- Adding Triggers
- Setting Advanced Trigger Controls
- Changing Trigger Execution Order
- Removing Triggers
- Refining When a Trigger Executes
- Adding a Trigger in the Body of a Method

Adding Triggers

A trigger is composed of three pieces of information: method, event, and action. The method is the method on which you want to set a trigger. The event controls when the

action takes place, that is, on entry to or exit from the method. The action corresponds to an action that you would normally perform from the JProbe Console.

Caution Triggers are global, which means that they are not local to methods or even threads. Each trigger occurs independently when the method is encountered and affects the whole program.

The procedure for adding triggers is the same for all the analysis tools, however the trigger actions are a little different among the tools.

Note You can add triggers during the wizard on the Specify JProbe Analysis screen, or when editing the configuration on the **Analysis Type** tab.

To add a trigger:

1. On the screen/tab where you define the analysis type, click the **Automation** tab, and click **Add Trigger**.

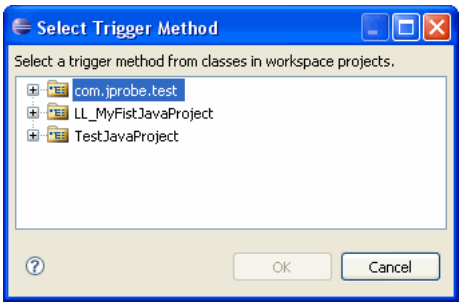
   The New Trigger Definition dialog box appears.The upper part of this dialog box is the same for all types of analysis, but the lower part contains different options, depending on which type of analysis you are performing and/or the action that you choose.

2. Do one of the following:

   - Select a trigger method from classes defined in your workspace projects:

   a. Click the browse button beside the **Trigger On Method** text box.

      The Select Trigger Method dialog box appears.



   b. Navigate to and select a class.

   c. Double-click the method you want.

- Enter the package, class, and method (no signature) in the cell. Wild cards cannot be used to specify the whole method name. For example: `*.*.methodD()` and `myPackage.MyClass.set*()` are valid, but `myPackage.MyClass.*()` or `*.*.*()` are not valid.

The checkmark at the beginning of the row indicates that the trigger is enabled. A default event and action are applied to the method.

3 To set when the trigger fires, select an option from the **Event** list.

4 To set the action to occur when the trigger fires, select an option from the **Action** list. For more information about each action, see:

- "Triggers for Memory" on page 125
- "Triggers for Performance" on page 271
- "Triggers for Coverage" on page 371

5 Change the options in the lower part of the screen, if desired.

- For Memory analyses: see "Setting the Recording Level" on page 110
- For Performance analyses: see "Setting the Recording Level" on page 236
- For Coverage analyses: select *Data Recording Off* or *Data Recording On*.

6 Click **Save**.

If you want to refine the action of the trigger, you can click **Advanced Control** to open the Set Advanced Trigger Control dialog box. For details, see "Setting Advanced Trigger Controls" on page 427.

Setting Advanced Trigger Controls

The Advanced Trigger Controls options allow you to have more control over when and/or how often a trigger action is performed. The two types of triggers are Method Invocation Based Trigger and Time Based Trigger. The options for each are the same, with some differences in the results. For examples of how to use these options, see "Refining When a Trigger Executes" on page 429.

The following table summarizes the options available for selection and describes what actions are performed with each type of trigger.

Option	Method Invocation Based	Time Based
Default Control	Perform the trigger action every time the method is executed.	Perform the trigger action every time the method is executed.
Perform the trigger action only once	Perform the trigger action on the [n]th execution of the method, where n is: (choose n from the drop-down list).	Perform the trigger action the first time the method is executed [n] minutes after program start, where n is: (choose n from the drop-down list).
Delay performing the trigger action	Start performing the trigger action on the [n]th execution of the method and then perform the trigger action every time the method is executed, where n is: (choose n from the drop-down list).	Start performing the trigger action the first time the method is executed [n] minutes after program start and then perform the trigger action every time the method is executed, where n is: (choose n from the drop-down list).
Customize when the trigger action is performed	Start performing the trigger action on the [n]th execution of the method and then perform the trigger action on every [m]th execution of the method. Stop performing the trigger action when it has been performed [x] times or when the application ends. The values of n, m, and x are: (choose n, m, and x from the drop-down list or select the **When the application ends** check box instead of x.).	Start performing the trigger action the first time the method is executed [n] minutes after program start and then perform the trigger action the first time the method is executed after every [m] minutes. Stop performing the trigger action when [x] minutes has elapsed since the trigger first fired, or when the application ends. The values of n, m, and x are: (choose n, m, and x from the drop-down list or select the **When the application ends** check box instead of x.).

Changing Trigger Execution Order

When two triggers on the same method occur on the same event, the first one in the table is executed first. Otherwise, trigger order in the table is not important.

Task	Action	Response
Move a trigger	Click the button at the left of the trigger's row and drag the trigger to the new location.	The trigger is inserted where you drop it.
Insert an empty row between two triggers	Right-click the button at the left of the trigger's row and choose **Insert**.	The trigger is inserted above the chosen trigger row.

Removing Triggers

You can remove triggers in one of two ways. If you think you may use the trigger again, you can disable the trigger for the next session. Otherwise, you can delete the trigger from the trigger table.

Task	Action	Response
Disable a trigger	Clear the check box beside the trigger's row.	The trigger is ignored during the subsequent analysis.
Re-enable a trigger	Select the check box beside the trigger's row.	The trigger is applied during the subsequent analysis.
Delete a trigger	Right-click the button at the left of the trigger's row and choose **Delete**.	The row is removed from the table.

Refining When a Trigger Executes

With a basic trigger, the trigger executes every time the method is called. You can refine this behavior so that a trigger executes based on the number of times a method is called or based on the amount of time that has elapsed.

Example: Method-based trigger control

For example, say you want the trigger to become active on the fifth time the method is called, and then fire on every other call to the method until the trigger has executed four times. To achieve this result, you would choose **Customize when the trigger action is performed** and set the values as n=5, m=2, and x=4. The trigger would fire as follows:

- fire the 5th time the method executes
- fire the 7th time the method executes
- fire the 9th time the method executes
- fire the 11th time the method executes
- the trigger becomes inactive after it has fired four times (or when the application ends, whichever comes first)

Example: Time-based trigger control

For example, let's say that you want to wait ten minutes after the application starts and then have the trigger fire when the method is called. Let's also assume that, for the next twenty minutes, you want to wait at least two minutes, fire the trigger when the method is called, then repeat. To achieve this result, you would choose **Customize when the trigger action is performed** and set the values as n=10, m=2, and x=20. The trigger would fire as follows:

- wait 10 minutes from program start, then fire the first time the method executes
- wait 2 minutes after the method call, then fire when the method executes again
- repeat the wait/method call/fire cycle until 20 minutes have elapsed (or the application ends, whichever comes first)
- the trigger becomes inactive after 20 minutes, or when the application ends, whichever comes first

The procedure for adding a trigger control is the same for all the analysis tools.

To add a trigger control:

1. Choose a trigger and click **Advanced Control**.

   The name and type of trigger is displayed at the top of the dialog box.

2. Choose whether the trigger is invocation based or time based:
   - To execute the trigger based on the number of times the method is called, choose **Method Invocation Based Trigger**.
   - To execute the trigger based on elapsed time, choose **Time Based Trigger**.

3 Choose when and for how long the advanced trigger is active:

- To execute the trigger once after a specified number method calls or minutes, choose **Perform the trigger action only once** and specify the number of methods call/minutes.
- To execute the trigger after a specified number of method calls or minutes and every time the method is called thereafter, choose **Delay performing the trigger action** and specify the number of methods call/minutes.
- To execute the trigger at specified intervals for a set duration, choose **Customize when the trigger action is performed** and specify values for the start, the interval, and the end.
- To remove an advanced control from a trigger, choose **Default control**.

4 Click **OK**.

An icon in the row button for the trigger indicates that an advanced control is set.

Adding a Trigger in the Body of a Method

Normally triggers can only act on entry or exit to a method. If you need a trigger in the body of a method (this provides you with greater control on executing the method) you can add an empty method to use as a trigger point, and place a call to it anywhere in your method, under any conditions you can express in Java code. Then when using JProbe, set a trigger on the added method.

For example, if you have a method like:

```
public String createFile() {
    setupFile();
    String filename = createFileName();
    // Would like to take a snapshot here if filename is null
    writeHeader(filename);
}
```

You want to take a snapshot before `writeHeader()` is called, but only if filename is null. You would add the following method:

```
private void triggerPoint() {}
```

And change `createFile()` as follows:

```
public String createFile() {
    setupFile();
    String filename = createFileName();
```

```
    if (filename == null) {
        // Would like to take a snapshot here if
        // using JProbe and filename is null
        triggerPoint();
    }
    writeHeader(filename);
}
```

It is at this point that you set a trigger on entry to the `triggerPoint()` method using the procedure in "Adding Triggers" on page 425.

Creating and Managing Categories

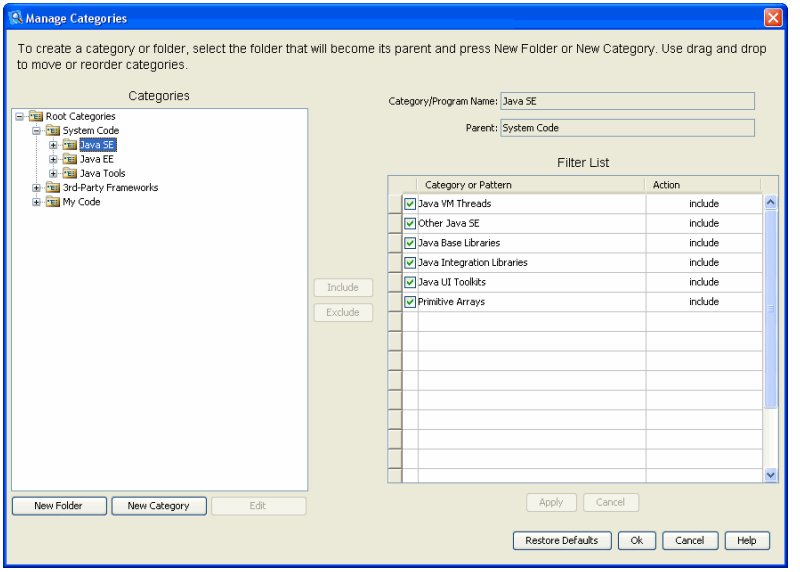
Categories provide a powerful tool for organizing the data that JProbe collects about your application. JProbe provides some preset categories, such as Java SE and Java EE, but you can also create your own. By default, JProbe creates a category for the code that you specify in the **My Code** tab of the Create/Edit Settings dialog box. Categories are used in the Memory analysis tool to group instances together.

Adding Categories

You add code to categories using filters. For information about filters, see "Syntax for Filters" on page 434.

To add a category:

1. Select **Tools > Manage Categories**.



2 Select the folder in which the category logically belongs and click **New Category**.

3 Enter a name in the Name field.

Tip It is recommended to choose a Category Name that is descriptive for the code you will be looking at. It is also recommended to specify unique Category Names, since overlapping patterns in Categories can be confusing, and you can use the same Category Name and filters in multiple configurations.

4 To add packages to this category:

- Click the browse button in the cell, navigate to a custom package, and click **Open**.
- Double-click the cell and enter the package name.

The package is displayed in the cell and the Action is set to `Include`.

5 To exclude some code from this category, click in an empty cell below the package/category row and do one of the following:

- To select a subpackage or class, click the cell browse button, navigate to the package or class, and click **Open**. Set the Action to `Exclude`.
- To select a method, click the browse button in the cell and navigate to the class containing the method. When you select the class, you can expand its list of

methods. Locate and double-click the desired method. Set the Action to `Exclude`.

- Enter the package, class, or method (no signature) in the cell. See the following subsection, Syntax for Filters. Set the Action to `Exclude`.

6 To add existing folders or categories to this category, choose the folder or category and click the **Include** button.

Note **Order is important.** Filters lower in the table override filters defined higher in the table. If you need to rearrange the filters, click and hold the row button for the filter you want to move and drag it to its new location. Right-click a row button to insert or delete a row.

7 Click **Apply**.

8 Repeat to create additional categories.

9 Click **OK**.

Syntax for Filters

Filters are case-sensitive and blank spaces are not allowed. You can use an asterisk (*) as a wild card; it matches any character, including the package separator. If you specify the code in a shorthand or partial form, JProbe automatically changes it to canonical form. The change may not be exactly what you desire, so be sure to verify it. JProbe does not check to see whether the filter actually matches anything in your code. For example, if you make a typing error in your package name, nothing in your code will match the filter and so the filter is not effective.

The following table contains common syntax and describes how JProbe interprets it:

If you type this:	Canonical form is this:	And filters are applied to:
`*`	`*.*.*()`	All methods in all classes in all packages (including the unnamed package)
`.*`	`.*.*()`	All methods in all classes in the unnamed package only
`C`	`*.C.*()`	All methods in classes named `C` in any package

If you type this:	Canonical form is this:	And filters are applied to:
`.C`	`.C.*()`	All methods in the class named `C` in the unnamed package only
`M()`	`*.*.M()`	Method `M` in all classes in all packages
`P.String*`	`P.String*.*()`	All methods in any class in the package `P` whose name starts with `String` and all methods in any class in a package or subpackage of `P.String*` (matches both `P.StringArt.draw()` and `P.Stringent.Check.English.spelling()`)
`P.*.C.do*()`	*no change*	Any method whose name starts with `do` in any class named `C` in any subpackage of `P`, except `P` directly (matches `P.SubPkg.C.doGet()`, but does not match `P.C.doNotMatch()`)
`Pre*`	`*.Pre*.*()`	All methods in all classes starting with `Pre` in any package, as well as any class in a subpackage whose name starts with `Pre` (matches both `Pkg.Prepare.m()` and `Pkg.Pretend.C.m()`)
`s*y`	`*.s*y.*()`	All methods in classes whose name ends with `y` and either starts with `s` or belongs to a subpackage that starts with `s` (matches both `java.rmi.server.RMISocketFactory`, `com.quest.say`, and `java.security.cert.X509CRLEntry`)
`foo.bar`	`foo.bar.*()`	All methods in the class `bar` in package `foo`. This is likely not what you wanted. See the next example.

If you type this:	Canonical form is this:	And filters are applied to:
`foo.bar.*`	`foo.bar.*.*()`	All methods in all classes in `foo.bar` or any of its subpackages
`foo.bar.String.t*e()`	*no change*	Methods whose names start with `t` and end in `e` in `foo.bar.String` (matches both `toUpperCase()` and `toLowerCase()`)

Adding Folders

To add a folder:

1. Select **Tools** > **Manage Categories**.
2. Select a folder and click **New Folder**.
3. Enter a name in the Name field.
4. Click **Apply**.
5. Repeat to create additional folders.
6. Click **OK**.

Editing Categories

To edit a category:

1. Select **Tools** > **Manage Categories**.
2. Select a category and click **Edit**.
3. Make changes to the filter list.

   Note A Category created automatically by JProbe should not be renamed (or at least a folder with the original name should be preserved), otherwise JProbe creates it again.
4. Click **Apply**.
5. Click **OK**.

Renaming Categories or Folders

To rename a category or folder:

1. Select **Tools > Manage Categories**.
2. Select a category, right-click, and select **Rename**.

   Note A Category created automatically by JProbe should not be renamed (or at least a folder with the original name should be preserved), otherwise JProbe creates it again.

3. Specify the new name and click **OK**.
4. Click **OK**.

Deleting Categories or Folders

To delete a category or folder:

1. Select **Tools > Manage Categories**.
2. Select a category, right-click, and select **Delete**.
3. Click **OK**.

23

Communicating Results

JProbe offers many ways to communicate your results to management, colleagues, and Quest Support (in case you encounter any issues).

This chapter presents how to create reports, print to PDF, save data to a CSV file, and save charts as images using JProbe. It also presents how you can view the analysis logs and the session history, and how to send a report bundle to Quest Support.

The following notes highlight the differences among these options:

- The content and layout of HTML reports is predefined. HTML reports are created using valid XHTML 1.0.
- Printing to PDF is WYSIWYG, that is, the output closely matches the current view. You can control what data is included and the order of that data by setting up the view before printing.
- CSV (comma-separated values) files contain raw data in a form suitable for import into other tools like spreadsheets. You can control what data is included and the order of that data by setting up the table before exporting to CSV.
- Image files enable you to save and print out the charts. You can set up the chart the way you want it before saving it as an image.

This chapter contains the following sections:

Generating HTML Reports 440
Printing Data to a PDF File 441
Exporting Table Data to a CSV File 442
Exporting Runtime Chart Data to a CSV File 443
Saving a Chart as an Image 443
Viewing the Execution Log 444
Viewing the Session History 445
Generating a Support Bundle 446

Generating HTML Reports

You can generate HTML reports based on many of the JProbe views. For example, you can generate a report on the Instances view or the Snapshot Browser view. The options in the Create Report dialog box differ depending on the analysis tool and the report being generated.

JProbe creates a report with a header that includes the report title, the version of JProbe, and the snapshot name, location, and timestamp. The content of HTML reports is predefined, however JProbe respects the table sort order, view filters, and type of units that you choose for the view. Data is always sorted in descending order.

HTML reports are created using valid XHTML 1.0, which means that they can be parsed and manipulated by XML tools.

Note You can generate many HTML reports using the command line tool `jpreport`. For more information, see the *JProbe Reference Guide*.

To create a report:

1. From a view that supports reports, click  **Export Data to HTML**.
2. Specify the fully qualified path to a file in the Output File field.
3. Set report options. For more information, see:
   - "Reporting Memory Analysis Results" on page 203
   - "Reporting Performance Analysis Results" on page 343
   - "Reporting Coverage Analysis Results" on page 399
4. To generate and display a sample of the report, click **Preview**.
5. To generate and save the report, click **Generate File**.

Printing Data to a PDF File

When printing to PDF, the output is WYSIWYG. You can set up the view the way you want to see it before printing. All visible data in the view is included in the output, even if you need to scroll to see it on screen. Adobe Reader is launched automatically from JProbe. You can view, save, and print the PDF from Adobe Reader.

To print the contents of a view to a PDF:

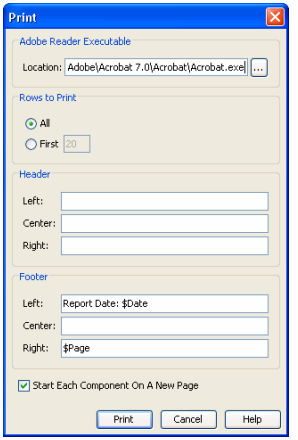
1. Open the view.
2. Set up the view:
   - For tables, you can set the table sort order, the columns shown, the order of those columns, and view filters (if available). You can also set the units in the Edit Options dialog.
   - For graphs that you want to print in black and white, ensure that the contrast among colors is strong enough.

     Tip Blue with light gray is a good choice for printing to black and white.
3. Click **Print**  on the toolbar, or click **File** > **Print**.
4. If this is the first time printing from JProbe, you are prompted to enter the location of the Adobe Reader executable.
5. For tables, you can choose the rows you want to print.
6. Specify the header and/or footer. Enter text or variables in the Left, Center, and/or Right fields, depending on where you want the text to appear in the output.

   You can use the following variables:

Variable	Description
`$Date`	The date and time that the report was created.
`$Class`	The class name of the application.
`$Application`	The fully qualified class name of the application.
`$Page`	The page number of the report.

7. To have components printed on separate pages, select the **Start Each Component On A New Page** check box.



8 Click **Print**.

Adobe Reader opens with the view results displayed. You can now save and print the results from Adobe Reader.

Exporting Table Data to a CSV File

When exporting to CSV (comma-separated values), all visible data is included in the output, even if you need to scroll to see it on screen. You can set up the view the way you want the data to be exported.

In the JProbe preferences, you can select whether or not headers are included in the exported data. For more information, see “Changing How JProbe Exports to a CSV File” on page 455.

To export a table of data:

1 Open the view.

2 Set up the view. You can set the table sort order, the columns shown, the order of those columns, and view filters (if available). You can also set the units in the **Edit Options** dialog.

3 To export the data, right-click in the table and choose **Export to CSV**.

4 Specify a location and file name.

5 Click **Save**.

Exporting Runtime Chart Data to a CSV File

When exporting a Memory or Performance runtime chart to CSV (comma-separated values), you can include heap data, non-heap data, or both. The heap data includes how memory is used in the heap, for example, you might see Eden Space, Survivor Space, and Tenured Gen values for each timestamp. The non-heap data includes other kinds of data, like Code Cache and Perm Gen, for each timestamp. Timestamps reflect clock time (not the elapsed time displayed in the chart) and are reported at five second intervals.

Note The Coverage Runtime view does not include a chart. Therefore this feature is unavailable.

In the JProbe preferences, you can choose whether or not headers are included in the data. For more information, see “Changing How JProbe Exports to a CSV File” on page 455.

To export a runtime view chart:

1. Open the Memory or Performance Runtime view.
2. To export the chart data, right-click in the chart and choose **Export to CSV**.
3. Specify a fully qualified file name.

   Tip You may want to add the .csv file extension to the file name for clarity.
4. Choose **Heap**, **Non-Heap**, or **Both** from the Pool Type drop-down list.
5. Click **OK**.

Saving a Chart as an Image

You can save a chart as a JPEG or PNG file and add that image to your reports. Set up the chart the way you want it before saving it as an image.

Note The Coverage views do not include charts. Therefore, this feature is unavailable.

To save a chart as an image:

1. Open the view containing the chart.
2. Set up the view.

3 Right-click in the chart and choose **Save Chart As** > <***image_type***>, where image type can be **JPEG** or **PNG**.

4 Specify a location and file name.

5 Click **Save**.

Viewing the Execution Log

When running a session, you can access the execution log from the Runtime view toolbar. The same information is also saved to every snapshot taken during the session.

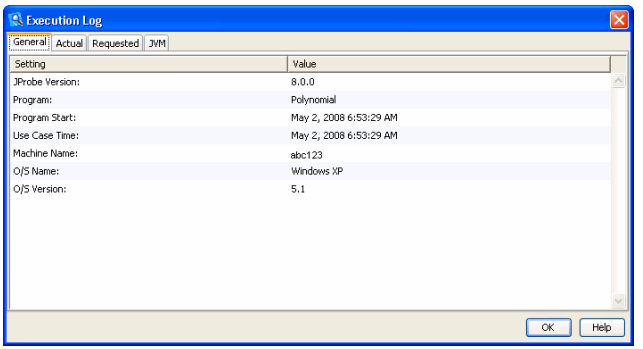
The following table summarizes the information in the execution log:

Tab Name	Description
General	Contains information about the JProbe version, the application, and the environment in which they run. For a snapshot, it also contains the snapshot name and file size.
Actual	Contains the options that were passed to the JProbe Analysis Engine, expressed as command line JProbe options. These may be different than the options that were requested in the configuration.
Requested	Contains the options that were set in the configuration, expressed as command line JProbe options.
JVM	Contains information about the Java executable running the application.

To view the execution log:

1 Do one of the following:

- In the Runtime view, click **Runtime Execution Log**  on the toolbar.
- In the Snapshots Navigator, right-click a snapshot and select **Execution Log**  .



2 Review the information.

3 Click **Close**.

Tip Errors and warnings that have occurred before users connected to the JProbe Analysis Engine are displayed on the system messages area in the runtime views, not on the Execution log.

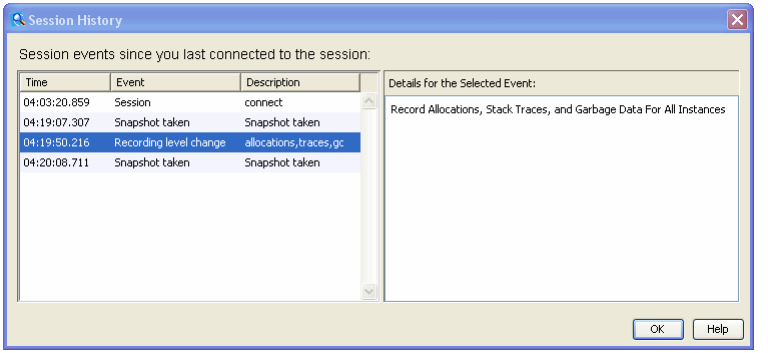
Viewing the Session History

When running a session, you can access the Session History from the Runtime Session views.

The Session History dialog presents events that have occurred since the Console has connected to the session (for example, the “start recording”, “stop recording”, “snapshot taken” events). A timestamp and a short description are provided in the left pane for each event. Additional details are presented in the right pane for the selected events. For example, the “start recording” event shows the details of the recording level and a list of filters (which may appear abbreviated, depending on their length), when applicable. Also, for example, the “snapshot taken” event shows the source of the request.

To view the session history:

1 In the Runtime Session view, click **View Session History**  on the toolbar.



2 Click **OK** to close the dialog box.

Generating a Support Bundle

When running a session, you can create a support bundle from the Runtime Session views.

To generate a support bundle:

1 In the Runtime Session view, click **Create Support Bundle**  on the toolbar.

JProbe automatically collects basic information about the session and generates a support bundle (as a ZIP file), for submission to Quest Support. The **Support Bundle Created** dialog box appears, indicating the location where the ZIP file was saved.

2 Optional: To copy to the clipboard the name of the bundle that was just created, click **Copy Bundle Name**.

3 Optional: To open the Quest Support Web page, click **Quest SupportLink**.

4 Click **OK** to close the dialog box.

24

Managing Snapshots

Snapshots contain the data collected about your application. If you have a snapshot, you can view the data on any JProbe Console or from Eclipse if you have installed the JProbe Plugins for Eclipse. You manage snapshots in the Snapshots Navigator.

Caution Snapshots are temporary; you need to save them before exiting or they are deleted.

This chapter presents how to manage the snapshots in your project. It contains the following sections:

Setting the Current Snapshot 448
Saving Snapshots 448
Loading Saved Snapshots 449
Unloading Snapshots 449
Saving Snapshots on Exit 450
Managing Snapshot Transfers 451

Setting the Current Snapshot

The current snapshot is the one supplying data to the views. By default, when the session ends, the last snapshot taken is the current snapshot. You can change the current snapshot to be one of the other snapshots listed in the Snapshots Navigator.

If you have more than one kind of snapshot loaded, you will have more than one current snapshot. For example, if you have both Coverage and Memory snapshots loaded, you have two current snapshots: one supplying data to the Memory views and one supplying data to the Coverage views.

To set the current snapshot:

1. In the Snapshots Navigator, double-click the snapshot.

   The snapshot name is changed to bold text to indicate that it is the snapshot providing data to the views. A default view may be displayed.

2. To open a view, right-click the snapshot and select **Open Snapshot**.

Saving Snapshots

You can save snapshots to review later. For example, you can compare results before and after a change to the program. When you save the snapshot, rename the file to something other than the default snapshot name or save it in a different directory.

Files with the default name in the target snapshot directory are deleted when you exit JProbe, unless you explicitly save them from the Save Snapshots dialog box. For more information, see “Saving Snapshots on Exit” on page 450.

To save a snapshot:

1. In the Snapshots Navigator, right-click a snapshot and select **Save Snapshot As**.
2. Navigate to a directory of your choice.
3. Type a name for the snapshot in the **File Name** field.
4. Click **Save**.

   The name of the snapshot is updated in the Snapshots Navigator.

Loading Saved Snapshots

You can load a saved snapshot into the JProbe Console.

To load a snapshot:

1. Click **File** > **Open Snapshot**.
2. Navigate to and select a snapshot file.

   Tip In Windows, each type of snapshot file has a specific icon associated with it.
3. Click **Open**.

   The snapshot is added to the list in the Snapshots Navigator.

Note If your snapshot is very large, it is recommended to use the `jpdumpreport` tool to generate a report for this view before importing it into JProbe. The `jpdumpreport` tool is an alternative for snapshots that are too big to be loaded, but, unlike `jpprepare`, `jpdumpreport` does not alter the snapshot and it does not make it load any faster. For details, see the *JProbe Reference Guide*.

Unloading Snapshots

Unloading a saved snapshot simply removes it from the Snapshots Navigator—you can open it again at any time. For example, if you take a snapshot, and then immediately take another that better represents the data, you can avoid confusion by unloading the previous snapshot.

Unloading an unsaved snapshot prompts you to save the snapshot. If you do not save it, the snapshot is deleted.

To unload a snapshot:

1. In the Snapshots Navigator, right-click a snapshot and select **Unload Snapshot**.
2. If you chose a temporary snapshot, you are prompted to save it.
   - To save the snapshot, click **Yes** and save the snapshot as usual.
   - To delete the snapshot, click **No**.
   - To leave the dialog box without unloading the snapshot, click **Cancel**.

   The snapshot is removed from the list in the Snapshots Navigator.

To unload all snapshots:

1. Choose **File > Unload All Snapshots**.
2. If some snapshots have not been saved, you are prompted to save them.
   - To save some unsaved snapshots, select the check boxes beside the snapshots and click **OK**. Save the snapshots as usual.
   - To save all unsaved snapshots, click **Select All** and click **OK**. Save the snapshots as usual.
   - To delete all unsaved snapshots, click **Clear All** and click **No**.
   - To leave the dialog box without unloading snapshots, click **Cancel**.

   All snapshots are unloaded and the Snapshots Navigator is empty.

Saving Snapshots on Exit

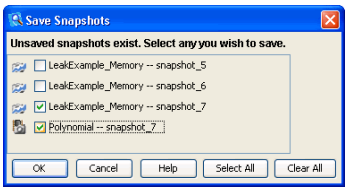
Snapshots are temporary; you need to save them before exiting or they are deleted. By default, you are prompted to save temporary snapshots when you exit JProbe.

Tip You can disable the prompt to save temporary snapshots. For more information, see "Controlling How JProbe Operates" on page 454.

To save multiple snapshots on exit:

1. Choose **File > Exit** to close JProbe.

   If there are unsaved snapshots, the Save Snapshots dialog box opens, listing all the temporary snapshots.
2. Choose the snapshots you want to save.



   - To save some unsaved snapshots, select the check boxes beside the snapshots and click **OK**. Save the snapshots as usual.
   - To save all unsaved snapshots, click **Select All** and click **OK**. Save the snapshots as usual.

- To delete all unsaved snapshots, click **Clear All** and click **OK**.
- To return to JProbe, click **Cancel**.

When the last snapshot is saved, JProbe closes. Any remaining temporary snapshots are deleted from the snapshot directory.

Managing Snapshot Transfers

Temporary snapshots are sent from the JProbe Analysis Engine to the JProbe Connection Manager and then to the JProbe Console. You can find out the status of the transfer.

To view the status of snapshot transfers:

1. Choose **Session > Snapshot Transfer Status**.
2. To stop downloading the current snapshot, click **Cancel**.
3. To avoid downloading pending snapshots, select the check boxes for the snapshots and click **Remove Checked Selections**.

25

Setting Preferences

This chapter presents how to set preferences that affect how JProbe operates and displays data.

For preference settings specific to an analysis tool, see:

- “Setting Preferences for Memory Views” on page 202
- “Setting Preferences for Performance Views” on page 342
- “Setting Preferences for Coverage Views” on page 390

This chapter contains the following sections:

Controlling How JProbe Operates 454
Changing How JProbe Exports to a CSV File 455
Hiding Package Names and Method Signatures 455
Specifying Source File Locations 455
Configuring Memory Pools 456

Controlling How JProbe Operates

This section describes how to change the default operating settings, such as hiding the welcome pages and disabling warnings.

To change how JProbe operates:

1. Click **Options**  on the toolbar, or select **Tools > Options** from the menu bar.

   By default, **General** options are displayed.

2. Do any of the following:

   - To show (or hide) the Welcome pages, select (or clear) the **Show Welcome Pages** check box.
   - To turn ON (OFF) the JProbe warning about unsaved snapshots when you exit, select (or clear) the **Warn on Unsaved Snapshots** check box.

     Caution When this options is disabled, temporary snapshots are deleted when JProbe exits.

   - To turn ON (OFF) the JProbe warning about comparing snapshots from different applications, select (or clear) the **Warn when Baseline and Snapshot to Compare are from Different Applications** check box.
   - To change where the JProbe Console stores the temporary files it requires while operating, specify the fully qualified path to the directory in the **Directory** field.

     Note The JProbe Analysis Engine and other JProbe components do not honor this setting.

   - To change the Web browser that JProbe uses for previews of reports, select **User Specified**. In the **Executable** and **Options** fields, specify the fully qualified path to the executable and any launch options, respectively.

3. Click **OK**.

Changing How JProbe Exports to a CSV File

You can select whether or not headers are included in the exported data.

To change how JProbe exports to a CSV file:

1. Click **Options** on the toolbar, or select **Tools > Options** from the menu bar.
2. Click **CSV Export**.
3. To include column headings in the exported data, select the **Export Column Headings** check box.
4. Click **OK**.

Hiding Package Names and Method Signatures

By default, wherever a class name appears in a view, the package name is prepended to it. When a method is identified, the method signature is appended. This section describes how to change the default data display settings.

To hide package names or method signatures:

1. Click **Options** on the toolbar, or select **Tools > Options** from the menu bar.
2. Click **Data Display**.
3. Do any of the following:
   - To hide method signatures, clear the **Show Method Signatures** check box.
   - To hide package names, clear the **Show Package Names** check box.
4. Click **OK**.

Specifying Source File Locations

You can tell JProbe where to look for source files by listing the directories containing your source code. The directories are searched from the top of the list to the bottom.

To specify the locations of your source files:

1. Click **Options** on the toolbar, or select **Tools > Options** from the menu bar.

2 Click **Source Files**.

3 Click an empty row and specify the fully qualified path to the root directory of your source file. For example, for class file `com.acme.Foo` where the source is in *c:/mysourcefiles/main/com/acme/Foo.java*, specify *c:/mysourcefiles/main*.

4 Enter additional directories, one per row.

5 To manage the list of directories:

- To move a directory earlier or later in the search order, choose the row and click **Move Up** or click **Move Down**.
- To insert an empty row above a row, choose the row and click **Insert**.
- To delete a directory, choose the directory and click **Delete**.

6 Click **OK**.

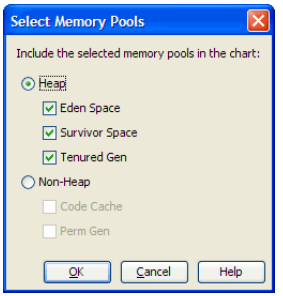
Configuring Memory Pools

A memory or performance chart can display heap or non-heap data. You can configure the type of data included in the chart by configuring the memory pools in the Runtime Summary view.

If you want to change the type of memory pool that is displayed in a memory chart:

1 Click **Configure Pools** Configure Pools... .

The *Select Memory Pools* dialog box appears.



2 Select **Heap** or **Non-Heap**.

Note These two options are mutually exclusive.

3 Select the check box beside each pool that must be displayed in the chart and clear the rest of the check boxes.

4 Click **OK**.

The chart refreshes to display the selected memory pools.

26

Running Remote Sessions

JProbe runs your program with the JVM you specify and listens for local or remote TCP/IP connections from the JProbe Console. JProbe uses a Connection Manager to act as a proxy for connections to both local and remote sessions. The Connection Manager simplifies the procedure for connecting the Console to the Analysis Engine and enables you to attach and detach the Console to and from a running session using the toolbar buttons.

In a remote session, there are two installations of JProbe: one on the server and one on the workstation. You set up and start the JProbe session on the server where your application runs, then you connect to the session from a workstation. For example, with a Web application, you configure a session to analyze Java Server Pages executing on a server, but you can sit at your own workstation to run the client browser and interact with the session.

Note Usually you run a remote session to analyze Java EE applications. However, you use the same procedure to set up a standalone application that you want to monitor from a remote computer.

This chapter describes how to run remote JProbe sessions. It contains the following section:

Starting and Viewing a Remote Session 460

Starting and Viewing a Remote Session

The following procedure assumes that you have a session settings (JPL) file and a startup script (BAT/CMD/SH) file that were either created on your local machine and uploaded to the server or created on the server machine using the JProbe Console.

Note If you created the JPL file on your local machine, ensure that all server paths are correct for the remote server environment.

If you are analyzing a Java EE application on an application server, this procedure assumes that you have created an application server integration on the server. For more information on using the command line to start Java SE and Java EE applications, see the *JProbe Reference Guide*.

To start and view a remote session:

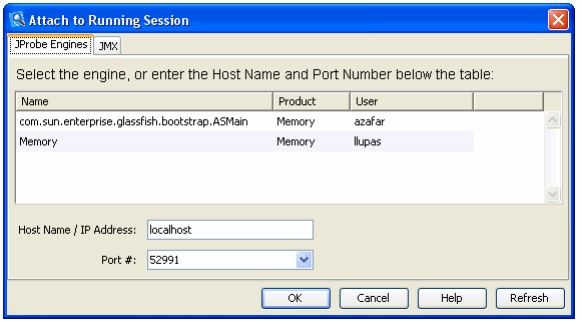
1. Connect to the server via a command window.
2. Run the startup script that you created in the JProbe Console.

   JProbe starts your program on your application server. You should see a message like the following:

   ```
   JProbe Connection Manager is available for Console
   connections on port 52991
   ```

   Note The default port number is 52991. If this port is unavailable, or if you have specified a different port in your JPL file arguments or on the command line, the port number is different.
3. Start JProbe on your local workstation.
4. Click **Attach to Session**  on the toolbar.

   The **Attach to Running Session** dialog box opens.



5 Enter the host name or IP address of the remote server and the port number that the Connection Manager is waiting on, and click **OK**.

The Console connects to the running session on the server. You can now view the session as you would a local session. For details about viewing data in the JProbe Console, see the other sections of this guide.

27

JProbe License Options

This chapter provides information about the license server that allows you to run JProbe sessions.

Before you can use JProbe, you must provide a valid license file. JProbe is unlocked for purchase or evaluation, depending on the license file you provide.

JProbe has four types of licensing models available: node locked, per seat, concurrent, and enterprise. For more information, review the JProbe licensing options on the Quest Web site at: *http://www.quest.com/jprobe/licensing_info.aspx.*

This chapter contains the following sections:

Node Locked Licensing 464
Per Seat Licensing 468
Concurrent Licensing 469
Enterprise Licensing 472

Node Locked Licensing

A node locked license allows JProbe to run on a predefined set of computers. The standard license is for a single developer. The license file contains the licensed host name and operating system, the type of license (purchase or evaluation), and the JProbe name, version, and license number. For evaluation (or leased) software, it also contains the evaluation expiration date.

Requesting Node Locked Licenses

You can obtain a node locked license while installing JProbe, or you can skip this step and request a license when you launch JProbe. If no license is installed, JProbe checks its root directory for a valid license on startup and adds it automatically, if found. Alternatively, you can go to the Quest Software SupportLink web site and request your license.

Requesting a License During Installation or from Within JProbe

This procedure assumes that you are working from the computer where your application and JProbe are installed, and that this computer has a graphical display and a connection to the Internet. When you request a license file, your hostname and number of CPUs is sent to Quest Software. This information is required for the license file.

Note If the computer you want to license does not have Internet access, or if you want to request a license for a different hostname, you can request a license through SupportLink. For more information, see “Requesting a License via SupportLink” on page 465.

If you want to run remote sessions, specify the second host name when the License Details web site is displayed. The second host can only run the JProbe Console for the product, not the JProbe Analysis Engine. After you receive the license file, you register the same license file on both computers.

To request a node locked license:

1. On the computer running your application and the JProbe Analysis Engine, do one of the following:
   - To request the license while installing JProbe, select the **Node Locked / Per Seat Licenses** tab in the Licensing Setup installation screen.

     Note This option is available only if you choose to run a Custom installation.

- To request the license after installation, launch JProbe and select the **Node Locked/ Per Seat Licenses** tab in the JProbe Licensing dialog box.

2 Click **Obtain License**.

You are advised that your hostname and number of CPUs will be sent to Quest Software. If you agree to transfer this information, continue with this procedure. Otherwise, see "Requesting a License via SupportLink" on page 465.

3 Click **OK**.

Your default browser opens. If the Quest Software license page does not display, ensure that cookies are enabled in your browser and try again.

4 Follow the instructions on the web site to request your license.

- If you are not subscribed, complete the subscription details.
- If someone else in your organization has subscribed with this license number, you may not be able to license JProbe on your computer.
- If you want to run remote sessions, be sure to add the host name of the second computer when prompted.

Tip To find the host name of a computer, enter `hostname` in a command prompt or xterm session.

5 After you complete the instructions on the Web site, your JProbe license file is sent to your email address. Save the license file to your computer.

6 Now you can register the node locked license. See "Registering a License During Installation or from Within JProbe" on page 466.

Requesting a License via SupportLink

You can request a JProbe license file using any computer that has Internet access. Go to the Get JProbe License page (URL below) and follow the online instructions. If the person using this copy of JProbe has not subscribed, you need to complete the subscription details before you can receive a license file.

Alternatively, instead of following this procedure, you can contact Quest's technical support to request a license file.

To request a node locked license:

1 Go to SupportLink at *http://support.quest.com*.

2 Select **License Keys**.

3 Follow the instructions on the web site to request your license.

- If you are not subscribed, complete the subscription details.
- If someone else in your organization has subscribed with this license number, you may not be able to license JProbe on your computer.
- When prompted, enter the host name of the computer running your application.
- If you want to run remote sessions, add the host name of the second computer from which you want to monitor JProbe sessions and review the collected data.

  Tip To find the host name of a computer, enter `hostname` in a command prompt or xterm session.

4. After you complete the instructions on the Web site, your JProbe license file is mailed to the subscribed person's email address. Save the license file to your computer.
5. Now you can register the node locked license. See "Registering a License During Installation or from Within JProbe" on page 466.

Registering Node Locked Licenses

You can register a node locked license while installing JProbe, or you can skip this step and register a license when you launch JProbe. Alternatively, you can run a command line tool to register JProbe.

Registering a License During Installation or from Within JProbe

This procedure walks you through registering a license file on the computer or computers named in the license file. It assumes that the computer has a graphical display. If not, a command line utility is also available. For more information, see "Registering Licenses from the Command Line" on page 468.

To register node locked licenses:

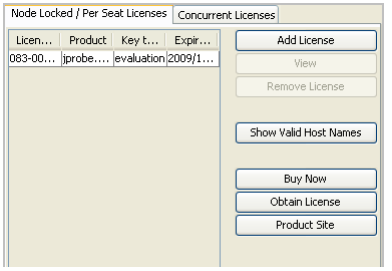
1. On the computer running your application and the JProbe Analysis Engine, do one of the following:
   - To register the license while installing JProbe, select the **Node Locked / Per Seat Licenses** tab in the Licensing Setup installation screen.

     Note This option is available only if you choose to run a Custom installation.

   - To register the license after installation, launch JProbe and select the **Node Locked / Per Seat Licenses** tab in the JProbe Licensing dialog box.

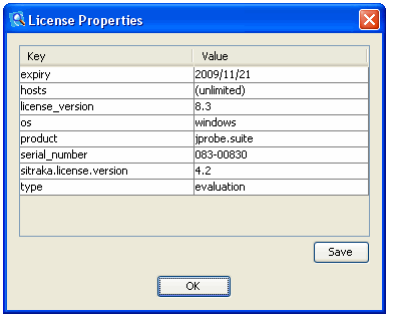
2 Click **Add License**, navigate to and select the saved license file, and click **Open**.

The license is installed to *JPROBE_HOME/config/jprobe.licenses*. Some of the details of the license are displayed in the table.



3 To view all the license details, select the license and click **View**.

The License Properties dialog opens with the details displayed.



4 Click **OK** to close the License Properties dialog box.

5 To save a backup copy of your license file in case you ever need to reinstall JProbe, click **Save**.

6 To end the license registration process, do one of the following:

- From the Licensing Setup installation screen, click **Next**.
- From the JProbe Licensing dialog box, click **Close**.

7 For a remote setup, go to the other computer that you specified when requesting the license file and repeat this procedure. If the license file is not accessible from the other computer, use FTP to put a copy of the license on the second computer.

8 You can now use JProbe.

Registering Licenses from the Command Line

You use the `jplicense` utility to register node locked licenses. You can also use this utility to list, view, replace, and delete licenses. For more information, see the *JProbe Reference Guide*.

Note You can request a JProbe license file using any computer that has Internet access. For more information, see "Requesting a License via SupportLink" on page 465.

To register node locked licenses from the command line:

1. On the computer running your application and JProbe, open a command prompt.
2. Navigate to the *JPROBE_HOME/bin* directory.
3. To add one or more node locked licenses, use the following syntax:
   - **Windows:** `jplicense add <license_file> [<license_file>...]`
   - **UNIX:** `./jplicense add <license_file> [<license_file>...]`

   where *license_file* is the fully qualified path to the JProbe license file(s).
4. To extract a node locked license as a backup in case you need to reinstall JProbe, use the following syntax:
   - **Windows:** `jplicense extract <license_number> <license_file>`
   - **UNIX:** `./jplicense` extract <*license_number*> <*license_file*>

   where *license_number* is the license to extract and *license_file* is the fully qualified path to the file where you want to save the extracted license. To get a list of license numbers, use `jplicense list`.
5. You can now use JProbe.

Per Seat Licensing

Seat licensing allows one individual user to use JProbe for a designated platform. This option is not covered in this guide. For more information, contact your Sales representative.

Concurrent Licensing

Concurrent licensing offers flexibility in where JProbe is installed and who may use the product. Instead of having one node locked license per developer, you purchase a number of licenses that are shared by the entire team via the Quest Software License Server.

Note Concurrent licensing is not available with evaluation versions of JProbe.

When you launch JProbe, the product checks to see if a license is available from the License Server. If so, you can use the product; if not, you need to try again later. If developers consistently have trouble obtaining a license, you can increase the number of licenses by contacting Quest Software. For more information, see “Using Concurrent Licenses” on page 469.

Using Concurrent Licenses

Concurrent licensing means that multiple people can start simultaneous JProbe analysis sessions on the same application. Each concurrent license permits a developer to run a JProbe Analysis Engine and a JProbe Console. Two people cannot connect JProbe Consoles to the same instance of a JProbe analysis session; a separate instance of the application is run under JProbe for each concurrent user. If no license is installed, JProbe queries the local subnet for a valid license server on startup and adds it automatically, if found.

Concurrent licenses are managed automatically. If you attempt to use JProbe while all licenses are in use, you are notified. You need to wait for an available license. If this occurs frequently, you may want to contact Quest Software to increase your number of licenses.

The following procedures assume that you have completed these tasks:

	Task	Help on Task
1	Purchased JProbe for the operating system that your application runs on and asked for concurrent licensing.	To order, contact **sales@quest.com.**

	Task	Help on Task
2	Downloaded the Quest Software License Server from the Web page specified in the confirmation email that you received, and specified the host name and OS of the network server where it will run.	To download the License Server, follow the instructions in the confirmation email, the License Details Web page, and the License Server download Web page**.** Your license file is emailed to you.
3	Installed and started the License Server on the network server you specified.	To install the License Server, run `licensing <version>-platform`. For example, `questlicensing4.2.7-lnx.bin`. To start the License Server, follow the instructions in the License Server documentation.
4	Added the license file to the License Server via its Web interface. The license file was mailed to the email address provided when you downloaded the License Server.	To open the home page for the License Server, see the Readme file. To add a license, follow the on-screen instructions. **Tip** You can also launch the Web interface from the JProbe **Concurrent Licenses** tab by selecting **View License Usage**.
5	Installed JProbe on the computer that runs your application.	See *JProbe Installation Guide*.
6	Installed JProbe on each of the developers' workstations.	See *JProbe Installation Guide*.
7	Ensured that all computers have access to the server running the License Server.	For help with network issues, see your System Administrator.

Setting the License Server During Installation or from Within JProbe

This procedure assumes that you are working from the computer running your application and the JProbe Analysis Engine. If you do not have a graphical display, see "Setting the License Server from the Command Line" on page 471.

To set the license server for concurrent licensing:

1. On the computer running your application and the JProbe Analysis Engine, do one of the following:
   - To set the license server while installing JProbe, select the **Concurrent Licenses** tab in the Licensing Setup installation screen.

     Note This option is available only if you choose to run a Custom installation.
   - To set the license server after installation, launch JProbe and select the **Concurrent Licenses** tab in the JProbe Licensing dialog box.
2. Type the fully qualified host name or IP address of the computer running the License Server in the License Server Hostname field.
3. Click **Test Server**.

   A message tells you whether or not the computer is available.
4. To set the specified host as your license server, click **Set Server**.
5. To view your license and the licenses currently in use, click **View License Usage**.
   - The License Server home page opens in your default browser.
   - For details on each screen, click the **Help** link.
   - To close the License Server web interface, close your browser.
6. To end the license registration process, do one of the following:
   - From the Licensing Setup installation, click **Next**.
   - From the JProbe Licensing dialog box, click **Close**.
7. Repeat on each of the developer workstations.
8. You can run JProbe if a concurrent license available.

Setting the License Server from the Command Line

You can test and set the license server from the command line. For more information, see “Using Concurrent Licenses” on page 469.

To set the license server for concurrent licensing from the command line:

1. From a command prompt, navigate to the *JPROBE_HOME/bin* directory.
2. To test that the license server is available, enter:
   - **Windows:** `jplicense test <license_server> <port> <application>`
   - **UNIX:** `./jplicense test <license_server> <port> <application>`

where *license_server* is the name of the server running the Quest Software License Server, *port* is the port number that the license server listens on, and *application* is the licensing application name. For example: `jplicense test serve_lics 8133 licensing`

3 To set the license server, use the following syntax:

Windows: `jplicense server <license_server> <port> <application>`

UNIX: `./jplicense server <license_server> <port> <application>`

where *license_server* is the host running the Quest Software License Server, *port* is the port number that the license server listens on, and *application* is the licensing application name.

4 Repeat on each of the developer workstations.

5 You can run JProbe if a concurrent license is available.

Enterprise Licensing

Enterprise licensing allows your entire development team to use JProbe. This option is not covered in this guide. For more information, contact your Sales representative.

Index

A

about JProbe 16
about Quest Software 20
adding
  categories 432
  folders 436
  triggers 425
advanced trigger controls 427
algorithms
  inefficient 220
  measuring improvements 258
Allocations Detail report 206
Allocations Detail view 163
  interacting with 164
  opening the 163
  understanding the 163
Allocations report 206
Allocations view 160
  interacting with 161
  opening the 160
  understanding the 160
Analysis Engine 417
analysis types, specifying 45
analyzing, a program using Performance 218
Apache
  Geronimo 76, 77, 78
  Tomcat 79, 80
application filters 38
application servers 31
  Apache Geronimo 76, 77, 78
  Apache Tomcat 79, 80
  BEA Systems WebLogic 81
  BEA Systems WebLogic Portal 81
  GlassFish 87, 88, 89
  IBM WebSphere AS 83, 85
  IBM WebSphere AS Community Ed. 82
  IBM WebSphere Portal Server 84
  JBoss 86, 87
  open source 87, 88, 89
  Oracle AS 10g 90, 91
  Oracle OC4J 10g 93
  Oracle WebLogic 92
  runtime notes for 75
  SpringSource 94
  unsupported 64
architecture, understanding the 414
arguments, specifying for a Java application 33
assessing
  existing test suite 356
  overall program performance 259
attaching to running JVM 96
  local application 96
  remote application 97
attaching to session 72, 73
  no connection 95
automation
  Coverage 371
  Memory 126
  Performance 268

B

baseline snapshot
  saving a Performance 239
  selecting 338
basename, for snapshots 48
BEA Systems
  WebLogic 81
  WebLogic Portal 81
bottlenecks
  identifying 243
  narrowing your investigation 249
browser, changing 454
building, a Calling Context Tree 222

C

Call Graph
  customizing 177, 187, 323
  customizing fonts 324
  hotspots 247
  navigating, by hiding and showing nodes 320
  node types 297
  pruning subtrees 319
  threads and methods 249
  viewing calling relationships 298
Call Graph tab 298
Call Traces
  customizing 177, 187
Call Traces report 207
Call Traces view 121, 174
  allocated live instances 176
  interacting with 178
  opening the 175
  understanding the 175
Call Tree
  isolating 228
  pruning 228
  threads and methods 249
Call Tree tab 302
calling context sites 330
Calling Context Tree
  building 222
  filtering and encapsulation 225
  how JProbe builds 223
  pruning subtrees 231
calling methods, navigating
  in Performance 322
  in the Call Graph 322
  subgraphs 321
catch blocks
  color-coding in Coverage Source view 385
  filtering out data 391
categories
  adding 432
  creating and managing 432
  deleting 437
  editing 436
  renaming 437
CCT
  See *Calling Context Tree*
changing, trigger execution order 429
chart
  exporting data to a file 443
  Memory Pools 221
  saving as an image 443
child navigator 247
classpath, specifying for a Java application 34
clearing snapshots 449
collecting data
  on lines of code 255
  on methods 222
  using triggers 256
collection references, obsolete 109
color
  interpreting, in Coverage Source view 385
  setting, for Coverage Source view 392
command line, licensing 468
communicating results 439
compiling your program 415
concurrent license
  overview 469
  using 469

conditions, details 386
configuration
  copying a 59
  creating a new 23
  creating from a settings file 61
  definition 29
  deleting a 61
  editing a 60
  managing 59
  renaming a settings file 63
  resolving errors 61
  saving 49, 50
  saving and integrating 55
Configurations Navigator
  overview 419
configure JProbe
  Java EE application 24
  Java SE application 27
  to analyze an application 23
configuring Memory Pools 456
Connection Manager 417
Console 417
console does not connect 95
contacting Quest 21
Context Calling Tree
  collecting data 223
  recursive nodes 232
  view in the Performance Call Graph 224
copying a configuration 59
Coverage
  analysis options 368
  analysis strategies 355
  assessing an existing test suite 356
  bar charts, hiding 391
  catch blocks, filtering out 391
  code changes 357
  color of results, setting 392
  concepts 393
  creating a test suite 356
  determining 356
  filters 369
  general settings 368
  hits, in Coverage Session 375
  investigating results 363
  learning more about results 393
  misses
    in Snapshot Browser 378
    in Source view 385
  modifying a test suite 356
  percentages as bar charts, displaying 391
  recording controls 372
  reports 399
  running an analysis 359
  running test cases 361
  setting a basic analysis 360
  Snapshot Browser 378
  Source view 384
  strategies 355
  triggers 371
  troubleshooting tips 410
  views, setting preferences for 390
Coverage reports, defining filters for 405
Coverage Session
  exploring the 374
  interacting with 376
  toolbar 374
  understanding data 375
Coverage Session report
  Class and Method Summaries 400
  Class Summary 400
Coverage Snapshot Browser
  interacting with 382
  opening 378
  understanding data 378
Coverage Source view 388
  Coverage Bar 385
  interacting with 388
  opening the 384
  understanding data 385
CPU time 257
creating, a test suite 356
critical path, finding hotspots 244

CSV file, exporting to 442, 443
customizing
  Call Graph 323
  Call Traces view 177, 187
  graph, color scale gradation mapping 323
  Method Detail view 333

D

data collection
  basic metrics 240
  catch counts 240
  exception counts 240
  filters 241
  metrics 241
  threaded programs 241
deadlocks
  definition 261
  detecting 263
  identifying 260
  investigating 264
  Threads tab 264
deleting
  a configuration 61
  categories 437
  folders 437
detecting deadlocks 263
determining, coverage 356
Difference dialog
  for heap comparison 198
  for Performance 338
Difference report
  Memory 208
  Performance 347
displaying data in the Method Detail view 330
documentation
  core 19
  feedback 19
  suite 18
Dominating Instances report 205
Dominating Instances view 148
  filtering data 151
  interacting with 152
  opening the 148
  understanding the 149
dominators 169
dump, importing 129

E

EAR file 43
Edit Options dialog
  Coverage
    catch blocks, filtering out 391
    colors, setting 392
    percentages as bar charts, displaying 391
  CSV export options, setting 455
  general options, setting 454
  method signatures, hiding 455
editing
  a configuration 60
  categories 436
EJB
  class 411
  JAR file 43
elapsed time 257
elements, creating filters from 43
encapsulating data on methods 250
encapsulation
  filtering methods 250
  full option 226, 251
  grouped option 227, 252
  none option 228, 253
  on methods 250
  options 225
  policy 269
enterprise license 472
errors, resolving in configurations 61
examining garbage collection data 120
execution log, viewing 444
exporting data from views 442, 443

F

filtering
  Coverage reports 405
  Heap Data 105
filtering data
  Dominating Instances view 151
  Instances view 141
filters
  Coverage 369
  creating from elements 43
  data collection, performance session runtime tip 352
  defining for application code 38
  for data collection 422
  Memory 124
  Performance 267
filters (data collection)
  adding 422
  adding filters using My Code Packages 423
  execution order, changing 424
  impact on metrics 241
  removing 424
finding
  hotspots 243
  memory leaks 108
folders
  adding 436
  deleting 437
  renaming 437

G

`-g` compiler option 415
garbage collection
  collecting data for tuning 118
  examining data 120
  issues 117
  monitoring with the Runtime Summary view 118
  tuning 117
GC Data
  Memory 104, 134
  Performance 276
generating reports 440
generating, a support bundle 446
Geronimo 76, 77, 78
GlassFish 87, 88, 89

H

Heap Data 134
  filtering 105
  interpreting 105
  Memory 105
Heap Difference View, understanding data 199
heap dump
  HPROF, importing 129
  IBM JVM, importing 127
  importing 127
  Sun JVM 129
Heap Graph report 208
Heap Graph view 185
  interacting with 187
  opening the 185
  understanding the 186
hiding
  nodes in the Call Graph 320
  subgraphs in the Call Graph 321
hits, in Coverage Session 375
hotspots
  critical path 244
  definition 220
  finding 243
how JProbe works 416
HTML file, exporting to 442, 443

I

IBM
  WebSphere AS 83, 85
  WebSphere AS Community Ed. 82
  WebSphere Portal Server 84
identifying deadlocks 260
importing
  heap dumps 127

IBM HPROF files 129
IBM PHD files 127
Instance Detail report 207
Instance Detail view 166
instance analysis 185
Memory, interacting with 171
opening the 166
referrers and references 185
understanding the 167
instance list 167
Instances report 204
Instances view 138
filtering data 141
opening the 138
understanding the 139
integrating
configurations 55
interacting with 336, 388
Allocations Detail view 164
Allocations view 161
Call Traces view 178
Coverage Session 376
Dominating Instances view 152
Heap Graph view 187
Instance Detail view 171
Leak Doctor view 195
Memory Difference view 200
Memory Runtime Summary view 135
Memory Source view 197
Merged Allocation Points view 182
Performance Difference view 340
Performance Methods view 305
Performance Runtime Summary view 279
Performance Summary view 292
Pinned Instances view 157
interpreting
Heap Data 105
JProbe analysis results 414
Memory Pools 103
investigating
Coverage results 363
deadlocks 264
performance bottlenecks 248
isolating subtrees in the Call Graph 319

J

JAR file 43
Java application
application arguments, setting 33
class path, setting 34
JVM, setting 36
main class, setting 32
running 72
working directory, setting 32
Java EE
configuring an application 24
running under JProbe 73
supported application servers 31
Java executable, specifying 36
Java file locations, setting 455
Java heap, root set 108
Java options, specifying 37
Java SE
configuring an application 27
specifying an application 31
JBoss 86, 87
JDBC component instrumentation, for
Performance 267
JProbe Analysis Engine 417
JProbe community 21
JProbe Connection Manager 417
JProbe Console
background 417
Configurations Navigator 419
connecting to
application started without JProbe 96
Java application running with JProbe 72
Java EE application running with JProbe 73
local application started without JProbe 96
remote application started without JProbe 97
interface 419

Navigator area 419
no connection 95
Snapshots Navigator 420
View area 419
JProbe options, setting 47
JVM, specifying for a Java application 36

L

Leak Doctor view 190
interacting with 195
opening the 190
understanding the 192
license
options 463
requesting 465
licensing
concurrent 469
enterprise 472
node locked 464
per seat 468
Line Level filter 241
line-level data
-g compiler 415
improving 415
lines of code, collecting data on 255
lingering transitional references 110
listeners, obsolete 109
loading snapshots 449
loitering objects 108

M

main class, specifying for a Java application 32
managing
configurations 59
licenses 463
measuring, performance improvements 258
Memory
Allocations Detail view 163
Allocations view 160
analysis options 124
analysis strategies 101
analysis, performing a basic 112
automation 126
Call Traces view 174
customizing the Call Traces view 177, 187
Difference view 197, 199
Dominating Instances view 148
filters 124
GC Data 104, 134
Heap Data 105, 134
heap dump, importing 127
Heap Graph view 185
help messages 215
importing IBM Dumps 127, 129
Instance Detail view 166
Instances view 138
Leak Doctor view 190
Memory Pools 103, 133
Merged Allocation Points view 181
Pinned Instances view 155
recording data 102
reports 203
running an analysis 107
Runtime Summary view 102, 132
session runtime tips 214
Set Recording Level 110
Source view 196
triggers 125
troubleshooting tips 212
views, setting preferences for 202
Memory Difference view 197
interacting with 200
opening the 198
percentages 200
using filters with 200
values 199
memory leaks, finding 108
Memory Pools 103
chart 221
configuring 456
interpreting the graph 103

Memory 133
Performance 275
Memory Runtime Summary view
interacting with 135
understanding the 132
Memory Source view 196
interacting with 197
opening the 197
Merged Allocation Points report 208
Merged Allocation Points view 120, 181
interacting with 182
opening the 181
understanding the 181
Method Detail report 347
Method Detail view 325
calling context sites 330
customizing the 333
edges from parents area 328
edges to children 330
graph area 328
navigating the 332
opening the 326
relationship to Methods view 326
summary information 327
understanding the 326
method detail, Coverage Snapshot Browser 381
Method Level, filter 241
methods
collecting data on 222
encapsulating data on 250
in Call Graph and Call Tree 249
signatures, hiding 455
Methods List 305
Methods report 346
Methods view 297
Call Graph tab 298
Call Tree tab 302
customizing the Call Graph 323
editing the data set 318
Methods List 305
navigating the graph 320
metrics
data collection filters and 241
threaded programs and 241
misses
in Coverage Source view 385
in Snapshot Browser 378
modifying, a test suite 356
multiple filters, in Heap Difference View 200

N

Native filter 241
navigating the, Method Detail view 332
navigators
child 247
parent 247
node locked license
overview 464
registering 466
requesting 464
nodes, hiding in the Call Graph 320
non-switch conditions 386

O

objects
loitering 108
obsolete
collection references 109
listeners 109
opcodes, definitions 388
opening snapshots 449
opening the
Allocations Detail view 163
Allocations view 160
Call Traces view 175
Coverage Source view 384
Dominating Instances view 148
Heap Graph view 185
Instance Detail view 166
Instances view 138
Leak Doctor view 190

Memory Difference view 198
Memory Source view 197
Merged Allocation Points view 181
Method Detail view 326
Performance Difference view 338
Performance Methods view 297
Performance Source view 335
Performance Summary view 283
Pinned Instances view 155
options
for Performance analysis 266
JProbe 46
Oracle
AS 10g 90, 91
OC4J 10g 93
WebLogic 92
over-allocations
finding 117
reducing 119

P

parent, navigator 247
PDF, generating 441
per seat license 468
percentages
hiding bar charts 391
in the Heap Difference View 200
in the Snapshot Difference view 339
Performance
analysis
options 266
strategies 217
analyzing a program 218
assessing program 259
automation 268
changing the gradation of the color scale 323
collecting data on methods 222
cumulative time and recursion 233
customizing the Call Graph 323
data, understanding 222
Difference view 338
encapsulation 225, 269
features, how data is affected by using 225
filtering 225, 267
full encapsulation 226
GC Data 276
grouped encapsulation 227
help messages 353
identifying
deadlocks 260
performance bottlenecks 243
inefficient algorithms 220
investigating
deadlocks 264
performance bottlenecks 248
your results 243
isolating Call Trees 228
JDBC component instrumentation 267
measuring improvements 258
Memory Pools chart 221, 275
Method Detail view 325
Methods view 297
narrowing your investigation 249
no encapsulation 228
pruning Call Trees 228
public methods 270
recursive methods 232
reports 343
running
a basic analysis 235
a session 237
Runtime Summary view 274
saving baseline snapshots 239
session runtime tips 351
Set Recording Level 236
setting up JProbe for an analysis session 236
snapshot, saving a baseline 239
Source view 335
strategies for program efficiencies 218
Summary view 283
Threads 276

timing settings 266
triggers 271
troubleshooting tips 350
understanding data collection 240
undo and redo buttons 323
views, setting preferences for 342
Performance Difference view
interacting with 340
opening the 338
understanding the 339
Performance Methods view
interacting with 305
opening the 297
understanding the 297
Performance Runtime Summary view
interacting with 279
understanding the 274
Performance Source view
interacting with 336
opening the 335
Performance Summary view
interacting with 292
opening the 283
understanding the 285
performing a basic Memory analysis 112
Pinned Instances report 205
Pinned Instances view
interacting with 157
opening the 155
understanding the 155
port number, changing 48
portable heap dump, importing 127
preferences, setting 453
printing to PDF 441
profiling data
collecting data on methods 222
understanding 222
pruning
subtrees 231
subtrees from the Call Graph 319
public method profiling 270

Q

querying, a set of instances 170
Quest Support
requesting modules 67

R

recording
controls, Coverage 372
data, Memory 102
recursive nodes, cumulative time definition 233
redo and undo buttons, Performance 323
redoing navigate actions in the Call Graph 323
reducing, objects over-allocation 119
references 168
referrers 168
Instance Detail View 168
reference tree 168
registering, node locked licenses 466
remote sessions
running 459
starting 460
removing
snapshots 449
triggers 429
renaming
a settings file 63
categories 437
folders 437
reports
Allocations 206
Allocations Detail 206
Call Traces 207
Coverage 399
Coverage Session 400
creating 440
Dominating Instances 205
Heap Graph 208
Instance Detail 207
Instances 204
Memory 203

Memory Snapshot Difference 208
Merged Allocation Points 208
Method Detail 347
Methods 346
Performance 343
Performance Snapshot Difference 347
Pinned Instances 205
Runtime Summary
Memory 204
Performance 344
setting options
Memory 204
Performance 344
Snapshot Browser 401
Summary 345
requesting, node locked licenses 464
results, communicating 439
root
See *loitering objects*
running
Coverage analysis 359
Java application 72
Java EE application 73
Memory analysis 107
remote sessions 459
scripts 55
Runtime Instances, filtering data 105
Runtime Summary report
Memory 204
Performance 344
Runtime Summary view
Memory 102, 132
Performance 274

S

saving
chart as an image 443
configurations 49, 50, 55
snapshots 448
snapshots on exit 450
script, running 55
session history, viewing 445
set of instances, querying a 170
Set Recording Level
Memory 110
Performance 236
setting
advanced trigger controls 427
Coverage options 360
current snapshot 448
Memory report options 204
Performance analysis options 266
Performance report options 344
trigger execution time 429
settings file
creating a configuration from a 61
integrating 55
renaming 63
saving 49, 50, 55
showing
nodes in the Call Graph 320
parent subgraphs in the Call Graph 322
previous view of the Call Graph 323
subgraphs in the Call Graph 321
top children in the Call Graph 321
top parents in the Call Graph 322
snapshot 418
baseline for heap comparison 198
baseline, saving 239
basename 48
compared 198
IBM Dump 127, 129
IBM HPROF file 129
IBM PHD file 127
overview 418
Snapshot Browser 378
Snapshot Browser report 401
Application Summary 402
Class Summary 403
Package Summary 403
Source Data 404

snapshots
loading 449
managing 447
saving 448
saving on exit 450
setting current 448
transfers, managing 451
unloading 449
Snapshots Navigator
adding snapshots to list 449
overview 420
source file locations, specifying 455
SpringSource, dm Server 94
stack references, stalled 110
stalled stack references 110
starting, a remote session 460
strategies, achieving program efficiency 218
subgraphs in the Call Graph, navigating 321
subtrees
isolating in the Call Graph 319
pruning, from the Call Graph 319
Summary report 345
Summary view 283
support 21
support bundle, generating 446
SupportLink, requesting a license 465
switch conditions 386

T

tables, exporting data to a file 442
tabs
Call Graph 298
Call Tree 302
GC Data
Memory 134
Performance 276
Heap Data 134
Memory Pools
Memory 133
Performance 275
Threads 276
technical support 21
temporary files location, changing 454
test cases, running Coverage 361
test suite
creating 356
modifying 356
text conventions 20
threaded programs and performance metrics 241
threads
in Call Graph and Call Tree 249
Threads tab 276
time, measuring 257
Tomcat 79, 80
traces 170
transferring snapshots 451
transitional references, lingering 110
triggers
adding 425
advanced controls 427
collecting data 256
collecting data using 256
Coverage 371
execution order, changing 429
managing 425
Memory 125
Performance 271
removing 429
setting execution time 429
within the body of a method 431
troubleshooting
Coverage analysis results 409
JProbe Console does not connect 95
Memory analysis results 211
Performance analysis results 349

U

understanding the
Allocations Detail view 163
Allocations view 160

Call Traces view 175
Dominating Instances view 149
garbage collection 117
Heap Graph view 186
Instance Detail view 167
Instances view 139
Leak Doctor view 192
Memory Difference view 199
Merged Allocation Points view 181
Method Detail view 326
Performance Difference view 339
Performance Methods view 297
Performance Summary view 285
Pinned Instances view 155
Runtime Summary view
Memory 132
Performance 274
undoing
navigate actions in the Call Graph 323
unloading snapshots 449
unsaved snapshot warning, disabling 454
unsupported application server
creating a configuration for 65
modifying a startup script 68
requesting modules from Quest Support 67
use cases, selecting 414

V

viewing
execution log 444
session history 445
views
Allocations 160
Allocations Detail 163
Call Traces 121, 174
Coverage Session 374
Coverage Source 384
Dominating Instances 148
for Coverage analysis 373
for Memory analysis 131
for Performance analysis 273
Heap Graph 185
Instance Detail 166
Instances 138
Leak Doctor 190
Memory Difference 197
Memory Source 196
Merged Allocation Points 120, 181
Method Detail 325
Methods 297
Performance Difference 338
Performance Source 335
Pinned Instances 155
printing 441
Runtime Summary
Memory 132
Performance 274
Summary 283

W

WAR file 43
warnings, resolving in configurations 61
Web browser, changing 454
WebLogic 81
WebLogic Portal 81
WebSphere AS 83, 85
WebSphere AS Community Ed. 82
WebSphere Portal Server 84
welcome pages, hiding 454
working directory, specifying for a Java application 32

Z

zero values
in the Snapshot Difference view 339